

**Daniel Danciu**

**George Mardale**

**ARTA PROGRAMĂRII ÎN JAVA**  
**vol II - Algoritmi și structuri de**  
**date**

# Cuprins

<b>9</b>	<b>Analiza eficienței algoritmilor</b>	<b>15</b>
9.1	Ce este analiza algoritmilor? . . . . .	16
9.2	Notăția asimptotică . . . . .	18
9.2.1	O notație pentru ordinul de mărime al timpului de execuție al unui algoritm . . . . .	19
9.3	Tehnici de analiza algoritmilor . . . . .	21
9.3.1	Sortarea prin selecție . . . . .	21
9.3.2	Sortarea prin inserție . . . . .	23
9.3.3	Turnurile din Hanoi . . . . .	24
9.4	Analiza algoritmilor recursivi . . . . .	25
9.4.1	Metoda iterației . . . . .	25
9.4.2	Inducția constructivă . . . . .	26
9.4.3	Recurențe liniare omogene . . . . .	26
9.4.4	Recurențe liniare neomogene . . . . .	28
9.4.5	Schimbarea variabilei . . . . .	31
9.5	Implementarea algoritmilor . . . . .	33
<b>10</b>	<b>Structuri de date</b>	<b>40</b>
10.1	Cum implementăm structurile de date? . . . . .	43
10.2	Stive . . . . .	45
10.3	Cozi . . . . .	51
10.4	Liste înlanțuite . . . . .	55
10.5	Arbori . . . . .	64
10.5.1	Noțiuni generale . . . . .	64
10.5.2	Arbori binari . . . . .	65
10.5.3	Arbori binari de căutare . . . . .	69
10.6	Tabele de repartizare . . . . .	83
10.6.1	Tratarea coliziunilor . . . . .	90

10.7	Cozi de prioritate . . . . .	92
10.7.1	Ansamble . . . . .	94
<b>11</b>	<b>Metoda Backtracking</b>	<b>108</b>
11.1	Prezentare generală . . . . .	109
11.2	Prezentarea metodei . . . . .	110
11.2.1	Atribuie și avansează . . . . .	114
11.2.2	Încercare eșuată . . . . .	114
11.2.3	Revenire . . . . .	115
11.2.4	Revenire după construirea unei soluții . . . . .	115
11.3	Implementarea metodei backtracking . . . . .	117
11.4	Probleme clasice rezolvabile prin backtracking . . . . .	119
11.4.1	Problema generării permutărilor . . . . .	119
11.4.2	Generarea aranjamentelor și a combinațiilor . . . . .	124
11.4.3	Problema damelor . . . . .	126
11.4.4	Problema colorării hărților . . . . .	127
<b>12</b>	<b>Divide et impera</b>	<b>137</b>
12.1	Introducere în recursivitate . . . . .	138
12.1.1	Funcții recursive . . . . .	138
12.1.2	Recursivitatea nu înseamnă recurență . . . . .	143
12.2	Prezentarea metodei Divide et Impera . . . . .	145
12.3	Căutare binară . . . . .	146
12.4	Sortarea prin interclasare (MergeSort) . . . . .	149
12.5	Sortarea rapidă (QuickSort) . . . . .	152
12.6	Expresii aritmetice . . . . .	157
<b>13</b>	<b>Algoritmi Greedy</b>	<b>167</b>
13.1	Problema spectacolelor (selectarea activităților) . . . . .	168
13.1.1	Demonstrarea corectitudinii algoritmului . . . . .	169
13.1.2	Soluția problemei spectacolelor . . . . .	171
13.2	Elemente ale strategiei Greedy . . . . .	173
13.2.1	Proprietatea de alegere Greedy . . . . .	176
13.2.2	Substructură optimă . . . . .	176
13.3	Minimizarea timpului mediu de așteptare . . . . .	177
13.4	Interclasarea optimă a mai multor șiruri ordonate . . . . .	180



<b>14 Programare dinamică</b>	<b>186</b>
14.1 Istoric și descriere . . . . .	187
14.2 Primii pași în programarea dinamică . . . . .	188
14.2.1 Probleme de recurență matematică tratate dinamic . . .	188
14.3 Fundamentare teoretică . . . . .	198
14.4 Principiul optimalității . . . . .	201
14.4.1 Metoda “înainte” și metoda “înapoi” . . . . .	204
14.4.2 Determinarea efectivă a soluției optime . . . . .	205
14.5 Înmulțirea unui șir de matrice . . . . .	208
14.6 Subșir comun de lungime maximă . . . . .	215
14.7 Distanța Levenstein . . . . .	220
<b>15 Metoda Branch &amp; bound</b>	<b>236</b>
15.1 Prezentare generală . . . . .	236
15.1.1 Fundamente teoretice . . . . .	237
15.2 Un exemplu: Puzzle cu 15 elemente . . . . .	240
15.2.1 Enunțul problemei . . . . .	240
15.2.2 Rezolvarea problemei . . . . .	241
<b>16 Metode de elaborare a algoritmilor (sinteză)</b>	<b>255</b>
16.1 Backtracking . . . . .	255
16.2 Divide et impera . . . . .	255
16.3 Greedy . . . . .	256
16.4 Programare dinamică . . . . .	256
16.5 Branch & bound . . . . .	256



# Introducere

Nu calculați capacitatea unui pod  
numărând persoanele care  
traversează acum râul înnot.

---

Auzită la o prezentare

Oricine a folosit cel puțin o dată Internetul sau a citit o revistă de specialitate în domeniul informaticii, a auzit cu siguranță cuvântul „Java”. Java reprezintă un limbaj de programare, creat de compania Sun Microsystems în anul 1995. Inițial, Java a fost gândit pentru a îmbunătăți conținutul paginilor web prin adăugarea unui conținut dinamic: animație, multimedia etc. În momentul lansării sale, Java a revoluționat Internetul, deoarece era prima tehnologie care oferea un astfel de conținut. Ulterior au apărut și alte tehnologii asemănătoare (cum ar fi Microsoft ActiveX sau Macromedia Shockwave<sup>1</sup>), dar Java și-a păstrat importanța deosebită pe care a dobândit-o în rândul programatorilor, în primul rând datorită facilităților pe care le oferea. Începând cu anul 1998, când a apărut versiunea 2 a limbajului (engl. Java 2 Platform), Java a fost extins, acoperind și alte direcții de dezvoltare: programarea aplicațiilor *enterprise* (aplicații de tip server), precum și a celor adresate dispozitivelor cu resurse limitate, cum ar fi telefoane mobile, *pager*-e sau PDA-uri<sup>2</sup>. Toate acestea au reprezentat facilități noi adăugate limbajului, care a păstrat însă și posibilitățile de a crea aplicații standard, de tipul aplicațiilor în linie de comandă sau aplicații bazate pe GUI<sup>3</sup>.

---

<sup>1</sup>Cei care utilizează mai des Internetul sunt probabil obișnuiți cu controale ActiveX sau cu animații flash în cadrul paginilor web.

<sup>2</sup>PDA = Personal Digital Assistant (mici dispozitive de calcul, de dimensiuni puțin mai mari decât ale unui telefon mobil, capabile să ofere facilități de agendă, dar și să ruleze aplicații într-un mod relativ asemănător cu cel al unui PC). La momentul actual există mai multe tipuri de PDA-uri: palm-uri, pocketPC-uri, etc.

<sup>3</sup>GUI = Graphical User Interface, interfață grafică de comunicare cu utilizatorul, cum sunt în general aplicațiile disponibile pe sistemul de operare Microsoft Windows.

Lansarea versiunii 2 a limbajului Java a fost o dovadă a succesului imens de care s-au bucurat versiunile anterioare ale limbajului, dar și a dezvoltării limbajului în sine, a evoluției sale ascendente din punct de vedere al facilităților pe care le oferă, cât și al performanțelor pe care le realizează.

## Cum este organizată această carte?

Având în vedere popularitatea extraordinară de care se bucură limbajul Java în cadrul programatorilor din întreaga lume, am considerat utilă scrierea unei lucrări în limba română care să fie accesibilă celor care doresc să învețe sau să aprofundeze acest limbaj. Ideea care a stat la baza realizării acestei cărți a fost aceea de a prezenta nu numai limbajul Java în sine, ci și modul în care se implementează algoritmii și structurile de date fundamentale în Java, elemente care sunt indispensabile oricărui programator cu pretenții. Așadar, cartea nu este destinată numai celor care doresc să acumuleze noțiuni despre limbajul Java în sine, ci și celor care intenționează să își aprofundeze și rafineze cunoștințele despre algoritmi și să își dezvolte un stil de programare elegant. Ca o consecință, am structurat cartea în două volume: prima volum (disponibil separat) este orientat spre prezentarea principalelor caracteristici ale limbajului Java, în timp ce volumul al doilea (cel de față) constituie o abordare a algoritmilor din perspectiva limbajului Java. Finalul primului volum cuprinde un grup de cinci anexe, care prezintă mai amănunțit anumite informații cu caracter mai special, deosebit de utile pentru cei care ajung să programeze în Java. Am ales această strategie deoarece a dobândi cunoștințe despre limbajul Java, fără a avea o imagine clară despre algoritmi, nu reprezintă un progres real pentru un programator. Iar scopul nostru este acela de a vă oferi posibilitatea să deveniți un programator pentru care limbajul Java și algoritmii să nu mai constituie o necunoscută.

Cele două volume cuprind numeroase soluții Java complete ale problemelor prezentate. Mai este necesară o remarcă: deseori am optat, atât în redactarea codului sursă cât și în prezentarea teoretică a limbajului sau a algoritmilor, pentru păstrarea terminologiei în limba engleză în defavoarea limbii române. Am luat această decizie, ținând cont de faptul că mulți termeni s-au consacrat în acest format, iar o eventuală traducere a lor le-ar fi denaturat înțelesul.

Primul volum al cărții cuprinde opt capitole:

**Capitolul 1** reprezintă o prezentare de ansamblu a tehnologiei Java. Capitolul debutează cu istoria limbajului, începând cu prima versiune și până la cea curentă. În continuare, sunt înfățișate câteva detalii despre implementările existente ale limbajului. Implementarea Java a firmei Sun este tratată separat, în detaliu, fiind și cea pe care s-au testat aplicațiile realizate pe parcursul cărții.

**Capitolul 2** este cel care dă startul prezentării propriu-zise a limbajului Java,

începând cu crearea și executarea unui program simplu. Sunt prezentate apoi tipurile primitive de date, constantele, declararea și inițializarea variabilelor, operatorii de bază, conversiile, instrucțiunile standard și metodele.

**Capitolul 3** este destinat referințelor și obiectelor. Sunt prezentate în detaliu noțiunile de referință, obiect, șiruri de caractere (String-uri) și de șiruri de elemente cu dimensiuni variabile.

**Capitolul 4** continuă prezentarea începută în capitolul anterior, prezentând modul în care se pot defini propriile clase în Java și cum se implementează conceptele fundamentale ale programării orientate pe obiecte.

**Capitolul 5** prezintă în detaliu un principiu esențial al programării orientate pe obiecte: *moștenirea*. Sunt prezentate de asemenea noțiuni adiacente cum ar fi cea de polimorfism, interfață, clasă interioară, identificare a tipurilor de date în faza de execuție (RTTI = Runtime Type Identification).

**Capitolul 6** este dedicat în întregime modului de tratare a excepțiilor în Java. Se prezintă tipurile de excepții existente în limbajul Java, cum se pot defini propriile tipuri de excepții, cum se pot prinde și trata excepțiile aruncate de o aplicație. Finalul capitolului este rezervat unei scurte liste de sugestii referitoare la utilizarea eficientă a excepțiilor.

**Capitolul 7** prezintă sistemul de intrare-ieșire (I/O) oferit de limbajul Java. Pe lângă operațiile standard realizate pe fluxurile de date (*stream*) și fișiere secvențiale, este prezentată și noțiunea de *colecție de resurse* (engl. resource bundles).

**Capitolul 8** este rezervat problemei delicate a firelor de execuție (*thread-uri*). Pornind cu informații simple despre firele de execuție, se continuă cu accesul concurrent la resurse, sincronizare, monitoare, coordonarea firelor de execuție, cititorul dobândind în final o imagine completă asupra sistemului de lucru pe mai multe fire de execuție, așa cum este el atât de elegant realizat în Java.

Cele opt capitole ale primului volum sunt urmate de un grup de anexe, care conțin multe informații utile programatorilor Java.

**Anexa A** constituie o listă cu editoarele și mediile integrate pentru dezvoltarea aplicațiilor Java, precum și un mic tutorial de realizare și executare a unei aplicații Java simple. Tot aici este prezentată și *ant*, o unealtă de foarte mare ajutor în compilarea și rularea aplicațiilor de dimensiuni mai mari.

**Anexa B** este dedicată convențiilor de scriere a programelor. Sunt prezentate principalele reguli de scriere a unor programe lizibile, conforme cu standardul stabilit de Sun Microsystems. Ultima parte a anexei este dedicată unei unelte foarte utile în documentarea aplicațiilor Java: *javadoc*.

**Anexa C** detaliază ideea de pachete, oferind informații despre pachete Java

predefinite și despre posibilitatea programatorului de a defini propriile sale pachete. Un accent deosebit se pune și pe prezentarea arhivelor `jar`.

**Anexa D** prezintă modul în care se pot realiza aplicații internaționalizate, prin care textele care apar într-o aplicație sunt traduse dintr-o limbă în alta, cu un efort minim de implementare. De asemenea, este prezentat și rolul colecțiilor de resurse (engl. *resource bundles*) în internaționalizarea aplicațiilor.

**Anexa E** reprezintă o listă de resurse disponibile programatorului Java, pornind de la *site*-ul Sun Microsystems și până la tutoriale, cărți, reviste online, liste de discuții disponibile pe Internet. Cu ajutorul acestora, un programator Java poate să își dezvolte aptitudinile de programare și să acumuleze mai multe cunoștințe despre limbajul Java.

Volumul de față, al doilea al cărții, este destinat prezentării algoritmilor. Independența algoritmilor relativ la un anumit limbaj de programare, face ca majoritatea programelor din această parte să fie realizate și în pseudocod, punctând totuși pentru fiecare în parte specificul implementării în Java.

**Capitolul 9** constituie primul capitol al celui de-al doilea volum și prezintă modalitatea prin care se poate realiza analiza eficienței unui algoritim. Notăția asimptotică, tehnicile de analiză a algoritmilor, algoritmi recursivi constituie principala direcție pe care se axează acest capitol.

**Capitolul 10** reprezintă o incursiune în cadrul structurilor de date utilizate cel mai frecvent în conceperea algoritmilor: stive, cozi, liste înlanțuite, arbori binari de căutare, tabele de repartizare și cozi de prioritate. Fiecare dintre aceste structuri beneficiază de o prezentare în detaliu, însoțită de o implementare Java în care se pune accent pe separarea interfeței structurii de date de implementarea acesteia.

**Capitolul 11** constituie startul unei suite de capitole dedicate metodelor fundamentale de elaborare a algoritmilor. Primul capitol din această suită este rezervat celei mai elementare metode: *backtracking*. După o analiză amănunțită a caracteristicilor acestei metode (cum ar fi cei patru pași standard în implementarea metodei: *atribuie și avansează, încercare eșuată, revenire, revenire după construirea unei soluții*), sunt prezentate câteva exemple de probleme clasice care admit rezolvare prin metoda *backtracking*: generarea permutărilor, a aranjamentelor și a combinațiilor, problema damelor, problema colorării hărților.

**Capitolul 12** prezintă o altă metodă de elaborare a algoritmilor: *divide et impera*. Prima parte a capitolului este rezervată prezentării unor noțiuni introductive despre recursivitate și recurență, absolut necesare înțelegerii modului în care funcționează această metodă. Analog capitolului 11, prezentarea propriu-zisă a metodei este însoțită de câteva exemple de probleme clasice rezolvabile prin această metodă: căutarea binară, sortarea prin interclasare (*mergesort*),

sortarea rapidă (*quicksort*), trecerea expresiilor aritmetice în formă poloneză postfixată.

**Capitolul 13** prezintă cea de-a treia metodă de elaborare a algoritmilor: *metoda Greedy*. Capitolul păstrează aceeași structură ca și cele precedente: sunt prezentate mai întâi elementele introductive ale metodei, urmate apoi de câteva exemple clasice de probleme rezolvabile prin această metodă: problema spectacolelor, minimizarea timpului de așteptare, interclasarea optimă a mai multor șiruri ordonate.

**Capitolul 14** este rezervat unei metode speciale de elaborare a algoritmilor: *programarea dinamică*, ce reprezintă probabil cea mai complexă metodă de elaborare a algoritmilor, punând deseori în dificultate și programatorii experimentați. Totuși avem credința că modul simplu și clar în care sunt prezentate noțiunile să spulbere mitul care înconjoară această metodă. Capitolul debutează cu o fundamentare teoretică a principalelor concepte întâlnite în cadrul metodei. Apoi, se continuă cu rezolvarea unor probleme de programare dinamică: înmulțirea unui șir de matrice, subșirul comun de lungime maximă, distanța Levenshtein etc.

**Capitolul 15** reprezintă ultimul capitol din seria celor dedicate metodelor de elaborare a algoritmilor. Metoda *branch and bound* este cea abordată în cadrul acestui capitol, prin intermediul unui exemplu clasic de problemă: jocul de *puzzle* cu 15 elemente.

**Capitolul 16** reprezintă o sinteză a metodelor de elaborare a algoritmilor care au fost tratate de-a lungul volumului al doilea, prezentând aspecte comune și diferențe între metode, precum și aria de aplicabilitate a fiecărei metode în parte.

Esența acestui volum o reprezintă metodele fundamentale de elaborare a algoritmilor împreună cu structurile de date cele mai uzuale, precum și modul în care acestea se particularizează pentru a fi implementate în limbajul Java.

Primul capitol al acestei părți introduce noțiuni esențiale referitoare la analiza eficienței algoritmilor, noțiuni care vor fi ulterior folosite pentru a evalua eficiența diferitelor operații pe structuri de date, precum și a algoritmilor prezentați. Următorul capitol introduce succesiv structurile de date cele mai uzuale, începând cu listele și încheind cu structuri ceva mai complexe cum ar fi arborii sau cozile de prioritate.

Restul capitolelor (de la capitolul 11 până la capitolul 15) prezintă pe rând metodele fundamentale de elaborare a algoritmilor care reprezintă niște tehnici cu caracter general prin care se poate rezolva o anumită clasă largă de probleme. Aceste metode nu sunt legate de un limbaj de programare anume, și din acest motiv mulți autori preferă să le trateze la modul general, descriind re-

zolvarea problemelor în pseudocod. În această lucrare am urmărit să prezentăm metodele de elaborare a algoritmilor la modul general, precum și specificul dat de implementarea rezolvărilor în limbajul Java.

Metodele de elaborare a algoritmilor prezentate în această parte sunt:

- **Backtracking:** se aplică problemelor a căror soluție se poate scrie sub formă de vector. Această metodă construiește vectorul soluție componentă cu componentă, cu eventuale reveniri asupra componentelor anterioare;
- **Greedy:** principiul acestei metode este asemănător cu cel de la *backtracking*, cu diferența că selectarea următorului pas se face pe baza unui criteriu local fără a se reveni asupra pașilor anteriori;
- **Divide et impera** (dezbină și cucerește): această metodă împarte problema originală în două sau mai multe subprobleme; subproblemele sunt împărțite la rândul lor în sub-subprobleme și așa mai departe până când se ajunge la subprobleme de dimensiune mică, a căror rezolvare este trivială. Se construiește apoi soluția problemei originale prin combinarea soluțiilor subproblemelor. Evident, nu orice problemă poate fi rezolvată în acest mod;
- **Programare dinamică:** aplicabilă doar problemelor de optimizare (sau care pot fi reformulate ca probleme de optimizare) care respectă așa-numitul *principiu al optimalității*;
- **Branch & bound:** termen care ar putea fi tradus cu aproximație prin "împarte și evaluează". Este o variantă a tehnicii *backtracking*, în care alegerea următorului pas nu se face la întâmplare, ci într-o anumită ordine dată de o evaluare locală a șanselor ca acel pas să conducă la o soluție.

Metodele de elaborare a algoritmilor au fost concepute ca niște tehnici cu caracter general aplicabile unei clase foarte largi de probleme de programare. Majoritatea problemelor de programare pot fi abordate cu una sau mai multe din aceste metode de elaborare a algoritmilor și, astfel, programatorul nu este întotdeauna nevoit să conceapă câte o metodă ad-hoc pentru fiecare problemă pe care o are de rezolvat. Programatorul trebuie să încadreze problema pe care o are de rezolvat în una dintre aceste metode de elaborare, după care particularizează acea metodă pentru problema concretă și alege structurile de date adecvate.

În linii mari, putem spune că rezolvarea unei probleme de programare presupune următorii pași:



1. Identificarea metodei de elaborare și a structurilor de date potrivite;
2. Particularizarea acestei tehnici pentru problema concretă.

În acest punct, tehnicile de programare se diferențiază: pentru unele metode (cum ar fi *backtracking*) trecerea de la forma generală la forma concretă este aproape algoritmică, nefiind necesar un mare efort de adaptare, în timp ce pentru altele (cum ar fi *programarea dinamică*) trecerea necesită un efort considerabil, dublat de ingeniozitate și o profundă stăpânire a metodei.

### Cui se adresează această carte?

Lucrarea de față nu se adresează începătorilor, ci persoanelor care stăpânesc deja, chiar și parțial, un limbaj de programare. Cititorii care au cunoștințe de programare în C și o minimă experiență de programare orientată pe obiecte vor găsi lucrarea ca fiind foarte ușor de parcurs. Nu sunt prezentate noțiuni elementare specifice limbajelor de programare cum ar fi funcții, parametri, instrucțiuni etc. Nu se presupune cunoașterea unor elemente legate de programarea orientată pe obiecte, deși existența lor poate facilita înțelegerea noțiunilor prezentate. De asemenea, cartea este foarte utilă și celor care doresc să aprofundeze studiul algoritmilor și modul în care anumite probleme clasice de programare pot fi implementate în Java.

### Pe Internet

Pentru comoditatea cititorilor, am decis să punem la dispoziția lor codul sursă complet al tuturor programelor prezentate pe parcursul celor două volume ale lucrării în cadrul unei pagini web concepută special pentru interacțiunea cu cititorii. De asemenea, pagina web a cărții va găzdui un forum unde cititorii vor putea oferi sugestii în vederea îmbunătățirii conținutului lucrării, vor putea schimba opinii în legătură cu diversele aspecte prezentate, adresa întrebări autorilor etc. Adresele la care veți găsi aceste informații sunt:

- <http://www.albastra.ro/carti/v178/>
- <http://www.danciu.ro/apj/>

### Mulțumiri

În încheiere, dorim să adresăm mulțumiri colegilor și prietenilor noștri care ne-au acordat ajutorul în realizarea acestei lucrări: Vlad Petric (care a avut o

contribuție esențială la structurarea capitolului 14), Alexandru Băluț (autor al anexei A), Iulia Tatomir (a parcurs și comentat cu multă răbdare de mai multe ori întreaga carte) și Raul Furnică (a parcurs și comentat capitolele mai delicate ale lucrării).

## 9. Analiza eficienței algoritmilor

Nu vă faceți griji pentru  
problemele pe care vi le pune  
matematica. Vă asigur că ale mele  
sunt cu mult mai mari.

---

Albert Einstein

În prima parte a cărții am examinat cum putem folosi programarea orientată pe obiecte pentru proiectarea și implementarea programelor profesionale. Au fost prezentate trăsăturile fundamentale ale limbajului Java, precum și facilități mai puțin cunoscute, dar foarte utile, cum ar fi mecanismul de reflectare (Reflection API) sau colecțiile de resurse. Totuși, aceasta reprezintă doar jumătate din problemă.

Calculatorul este folosit de obicei pentru a prelucra cantități mari de informație. Atunci când executăm un program cu date de intrare de dimensiuni mari, trebuie să fim siguri că vom obține rezultatul într-un timp rezonabil. Acest lucru este aproape întotdeauna independent de limbajul de programare folosit, ba chiar și de metodologia aplicată (cum ar fi programare orientată pe obiecte, sau programare procedurală).

Un algoritm este un set bine precizat de instrucțiuni pe care calculatorul le va executa pentru a rezolva o problemă. Odată ce am găsit un algoritm pentru o anumită problemă și am determinat că algoritmul este corect, pasul următor este de a determina cantitatea de resurse, cum ar fi timpul și cantitatea de memorie, pe care algoritmul le cere. Acest pas este numit analiza algoritmului. Un algoritm care are nevoie de câțiva *gigabytes* de memorie nu este bun de nimic pe calculatoarele existente la ora actuală, chiar dacă el este corect.

În acest capitol vom vedea:

- Cum putem estima timpul cerut de un algoritm (altfel spus, determinarea complexității algoritmului);

- Tehnici pentru reducerea drastică a timpului de execuție al unui algoritm;
- Un cadru matematic care descrie la un mod mai riguros timpul de execuție al algoritmilor.

## 9.1 Ce este analiza algoritmilor?

Cantitatea de timp pe care orice algoritm o cere pentru execuție depinde aproape întotdeauna de cantitatea de date de intrare pe care o procesează. Este de așteptat că sortarea a 10.000 de elemente să necesite mai mult timp decât sortarea a 10 elemente. Timpul de execuție al unui algoritm este astfel o funcție de dimensiunea datelor de intrare. Valoarea exactă a acestei funcții depinde de mai mulți factori, cum ar fi viteza calculatorului pe care rulează programul, calitatea compilatorului și, nu de puține ori, calitatea programului. Pentru un program dat, care rulează pe un anumit calculator, putem reprezenta grafic timpul de execuție. În **Figura 9.1** am realizat un astfel de grafic pentru patru programe. Curbele reprezintă patru funcții care sunt foarte des întâlnite în analiza algoritmilor: *liniară*,  $n \log n$ , *pătratică* și *cubică*. Dimensiunea datelor de intrare variază de la 1 la 100 de elemente, iar timpii de execuție de la 0 la 5 milisecunde. O privire rapidă asupra graficelor din **Figura 9.1** și **Figura 9.2** ne lămurește că ordinea preferințelor pentru timpii de execuție este liniar,  $n \log n$ , pătratic și cubic.

Să luăm ca exemplu descărcarea (download-area) unui fișier de pe Internet. Să presupunem că la început apare o întârziere de două secunde (pentru a stabili conexiunea), după care descărcarea se va face la 1.6 KB/sec. În această situație, dacă fișierul de adus are  $N$  kilobaiți, timpul de descărcare a fișierului este descris de formula  $T(N) = N/1.6 + 2$ . Aceasta este o funcție *liniară*. Se poate calcula ușor că descărcarea unui fișier de 80K va dura aproximativ 52 de secunde, în timp ce descărcarea unui fișier de două ori mai mare (160K) va dura 102 secunde, deci cam de două ori mai mult. Această proprietate, în care timpul este practic direct proporțional cu cantitatea de date de intrare, este specifică unui *algoritm liniar*, și constituie adeseori o situație ideală. Așa cum se vede din grafice, unele curbe neliniare pot conduce la timpi de execuție foarte mari.

În acest capitol vom analiza următoarele probleme: cu cât este mai bună o curbă în comparație cu o altă curbă, cum putem calcula pe care curbă se situează un anumit algoritm sau cum putem proiecta algoritmi care să nu se situeze pe curbele nefavorabile.

O funcție *cubică* este o funcție al cărei termen dominant este  $N^3$ , înmulțit cu o constantă. De exemplu,  $10N^3 + N^2 + 40N + 80$  este o funcție cubică.

Figura 9.1: Timpi de execuție pentru date de dimensiune mică

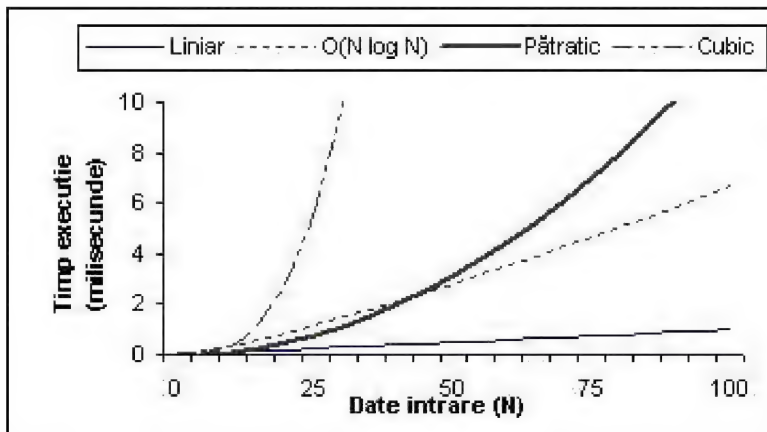
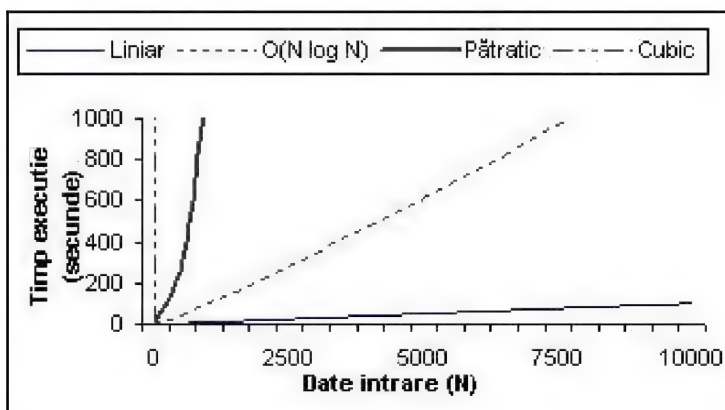


Figura 9.2: Timpi de execuție pentru date de intrare moderate



Similar, o funcție pătratică are termenul dominant  $N^2$  înmulțit cu o constantă, iar o funcție liniară are un termen dominant care este  $N$  înmulțit cu o constantă.

Oricare dintre cele trei funcții prezentate mai sus poate fi mai mică decât cealaltă într-un punct dat. Acesta este motivul pentru care nu ne interesează valorile efective ale timpilor de execuție, ci rata lor de creștere. Acest lucru este justificabil prin trei argumente. În primul rând, pentru funcțiile cubice, cum ar fi cea prezentată în **Figura 9.2**, atunci când  $N$  are valoarea 1000, valoarea funcției cubice este aproape complet determinată de valoarea termenului cubic. Funcția  $10N^3 + N^2 + 40N + 80$  are valoarea 10.001.040.080 pentru  $N = 1000$ , din care 10.000.000.000 se datorează termenului  $10N^3$ . Dacă am fi folosit doar termenul cubic pentru a estima valoarea funcției, ar fi rezultat o eroare de aproximativ 0.01%. Pentru un  $N$  suficient de mare, valoarea funcției este determinată aproape complet de termenul ei dominant (semnificația termenului *suficient de mare* depinde de funcția în cauză).

Un al doilea motiv pentru care măsurăm doar rata de creștere a funcțiilor este că valoarea exactă a constantei multiplicative pentru termenul dominant diferă de la un calculator la altul. De exemplu, calitatea compilatorului poate să influențeze destul de mult valoarea constantei. În al treilea rând, valorile mici pentru  $N$  sunt de obicei ne semnificative. Din **Figura 9.1** se observă că pentru  $N = 10$ , toți algoritmii se încheie în mai puțin de 3 ms. Diferența dintre cel mai bun și cel mai slab algoritm este mai mică decât un clipit de ochi.

Pentru a reprezenta rata de creștere a unui algoritm se folosește așa-numita notație asimptotică (engl. "Big-Oh notation"). De exemplu, rata de creștere pentru un algoritm pătratic este notată cu  $O(N^2)$ . Notația asimptotică ne permite să stabilim o ordine parțială între funcții prin compararea termenului lor dominant.

Vom dezvolta în acest capitol aparatul matematic necesar pentru analiza eficienței algoritmilor, urmărind ca această incursiune matematică să nu fie excesiv de formală. Vom arăta apoi, pe bază de exemple, cum poate fi analizat un algoritm. O atenție specială o vom acorda tehnicilor de analiză a algoritmilor recursivi.

## 9.2 Notația asimptotică

Notația asimptotică are rolul de a estima timpul de calcul necesar unui algoritm pentru a furniza rezultatul, funcție de dimensiunea datelor de intrare.

### 9.2.1 O notație pentru ordinul de mărime al timpului de execuție al unui algoritm

Fie  $\mathbf{N}$  mulțimea numerelor naturale,  $\mathbf{R}$  mulțimea numerelor reale. Fie  $f : \mathbf{N} \rightarrow [0, \infty)$  o funcție arbitrară. Definim mulțimea de funcții:

$$O(f) = \{t : \mathbf{N} \rightarrow [0, \infty) \mid \exists c > 0, \exists n_0 \in \mathbf{N}, \text{ astfel încât } \forall n \geq n_0 \text{ avem } t(n) \leq c * f(n)\}$$

Cu alte cuvinte,  $O(f)$  (se citește "ordinul lui  $f$ ") este mulțimea tuturor funcțiilor  $t$  mărginite superior de un multiplu real pozitiv al lui  $f$ , pentru valori suficient de mari ale argumentului  $n$ . Vom conveni să spunem că  $t$  este în ordinul lui  $f$  (sau, echivalent,  $t$  este în  $O(f)$ , sau  $t \in O(f)$ ) chiar și atunci când  $t(n)$  este negativ sau nedefinit pentru anumite valori  $n < n_0$ . În mod similar, vom vorbi despre ordinul lui  $f$  chiar și atunci când valoarea  $f(n)$  este negativă sau nedefinită pentru un număr finit de valori ale lui  $n$ ; în acest caz, vom alege  $n_0$  suficient de mare, astfel încât pentru  $n \geq n_0$  acest lucru să nu mai apară. De exemplu, vom vorbi despre ordinul lui  $n/\log n$ , chiar dacă pentru  $n=0$  și  $n=1$  funcția nu este definită. În loc de  $t \in O(f)$ , uneori este mai convenabil să folosim notația  $t(n) \in O(f(n))$ , subînțelegând aici că  $t(n)$  și  $f(n)$  sunt funcții.

Fie un algoritm dat și fie o funcție  $t : \mathbf{N} \rightarrow [0, \infty)$ , astfel încât o anumită implementare a algoritmului să necesite cel mult  $t(n)$  unități de timp pentru a rezolva un caz de mărime  $n$ , unde  $n \in \mathbf{N}$ . Principiul invarianței<sup>1</sup> ne asigură atunci că orice implementare a algoritmului necesită un timp în ordinul lui  $t$ . Cu alte cuvinte, acest algoritm necesită un timp în ordinul lui  $f$  pentru orice funcție  $f : \mathbf{N} \rightarrow [0, \infty)$  pentru care  $t \in O(f)$ . În particular avem relația:  $t \in O(t)$ . Vom căuta, în general, să găsim cea mai simplă funcție  $f$ , astfel încât  $t \in O(f)$ .

**Exemplu:** Fie funcția  $t(n) = 3n^2 - 9n + 13$ . Pentru  $n$  suficient de mare, vom avea relația  $t(n) \leq 4n^2$ . În consecință, luând  $c = 4$ , putem spune că  $t(n) \in O(n^2)$ . La fel de bine puteam să spunem că  $t(n) \in O(13n^2 - \sqrt{2}n + 12.5)$ , dar pe noi ne interesează să găsim o expresie cât mai simplă. Este adevărată și relația  $t(n) \in O(n^4)$  dar, așa cum vom vedea mai târziu, suntem interesați de a mărgini cât mai strâns ordinul de mărime al algoritmului, pentru a putea obiectiva cât mai bine durata sa de execuție.

Proprietățile de bază ale lui  $O(f)$  sunt date ca exerciții (1 - 5) și ar fi recomandabil să le studiați înainte de a trece mai departe.

<sup>1</sup> Acest principiu afirmă că două implementări diferite ale aceluiași algoritm nu diferă, ca eficiență, decât cel mult printr-o constantă multiplicativă.

Notația asimptotică definește o relație de ordine parțială între funcții și, prin urmare, între eficiența relativă a diferiților algoritmi care rezolvă o anumită problemă. Vom da în continuare o interpretare algebrică a notației asimptotice. Pentru oricare două funcții  $f, g : \mathbb{N} \rightarrow \mathbb{R}^*$  definim următoarea relație binară:  $f \leq g$  dacă  $O(f) \subseteq O(g)$ . Relația " $\leq$ " este o *relație de ordine parțială* (reflexivă, tranzitivă, antisimetrică) în mulțimea funcțiilor definite pe  $\mathbb{N}$  și cu valori în  $[0, \infty)$  (exercițiul 4). Definim și o *relație de echivalență*:  $f \equiv g$  dacă  $O(f) = O(g)$ . Prin această relație obținem clase de echivalență, o *clasă de echivalență* cuprinzând toate funcțiile care diferă între ele printr-o constantă multiplicativă. De exemplu,  $\lg n \equiv \ln n$  și avem o clasă de echivalență a funcțiilor logaritmice, pe care o notăm generic cu  $O(\log n)$ . Notând cu  $O(1)$  clasa de echivalență a algoritmilor cu timpul mărginit superior de o constantă (cum ar fi interschimbarea a două numere, sau maximum a trei elemente), ierarhia celor mai cunoscute clase de echivalență este:

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$$

Această ierarhie corespunde unei clasificări a algoritmilor după un criteriu al performanței. Pentru o problemă dată, dorim mereu să obținem un algoritm corespunzător unei clase aflate cât mai "de jos" (cu timp de execuție cât mai mic). Astfel, se consideră a fi o mare realizare dacă în locul unui algoritm exponențial găsim un algoritm polinomial.

Exercițiul 5 ne dă o metodă de simplificare a calculelor în care apare notația asimptotică. De exemplu:

$$\begin{aligned} n^3 + 4n^2 + 2n + 7 &\in O(n^3 + (4n^2 + 2n + 7)) = \\ &O(\max(n^3, 4n^2 + 2n + 7)) = O(n^3) \end{aligned}$$

Ultima egalitate este adevărată chiar dacă  $\max(n^3, 4n^2 + 2n + 7) \neq n^3$  pentru  $0 \leq n \leq 4$ , deoarece *notația asimptotică* se aplică doar pentru  $n$  suficient de mare. De asemenea,

$$\begin{aligned} n^3 - 3n^2 - n - 8 &\in O\left(\frac{n^3}{2} + \left(\frac{n^3}{2} - 3n^2 - n - 8\right)\right) = \\ &O\left(\max\left(\frac{n^3}{2}, \frac{n^3}{2} - 3n^2 - n - 8\right)\right) = O\left(\frac{n^3}{2}\right) = O(n^3) \end{aligned}$$

chiar dacă pentru  $0 \leq n \leq 6$  polinomul este negativ. Exercițiul 6 tratează cazul unui polinom oarecare.

Notația  $O(f)$  este folosită pentru a limita superior timpul necesar unui algoritm, măsurând eficiența (complexitatea computațională) a algoritmului respectiv. Uneori este interesant să estimăm și o limită inferioară a acestui timp. În acest scop, definim mulțimea:



$$\Omega(f) = \{t : \mathbb{N} \rightarrow [0, \infty) \mid \exists c > 0, \exists n_0 \in \mathbb{N}, \text{ astfel încât } \forall n \geq n_0 \text{ avem } t(n) \geq c * f(n)\}$$

Există o anumită dualitate între notațiile  $O(f)$  și  $\Omega(f)$ : pentru două funcții oarecare  $f, g : \mathbb{N} \rightarrow [0, \infty)$ , avem:

$$f \in O(g) \text{ dacă și numai dacă } g \in \Omega(f).$$

O estimare foarte precisă a timpului de execuție se obține atunci când timpul de execuție al unui algoritm este limitat atât inferior cât și superior de câte un multiplu real pozitiv al aceleiași funcții. În acest scop, introducem notația:

$$\Theta(f) = O(f) \cap \Omega(f)$$

numită *ordinul exact al lui f*. Pentru a compara ordinele a două funcții, notația  $\Theta$  nu este însă mai puternică decât notația  $O$ , în sensul că  $O(f) = O(g)$  este echivalent cu  $\Theta(f) = \Theta(g)$ .

Există situații în care timpul de execuție al unui algoritm depinde simultan de mai mulți parametri. Aceste situații sunt tipice pentru anumiți algoritmi care operează cu grafuri și la care timpul depinde atât de numărul de vârfuri cât și de numărul de muchii. Notația asimptotică se generalizează în mod natural și pentru funcții cu mai multe variabile. Astfel, pentru o funcție arbitrară  $f : \mathbb{N} \times \mathbb{N} \rightarrow [0, \infty)$  definim

$$O(f) = \{t : \mathbb{N} \times \mathbb{N} \rightarrow [0, \infty) \mid \exists c > 0, \exists n_0, m_0 \in \mathbb{N}, \text{ astfel încât } \forall m \geq m_0, \forall n \geq n_0 \text{ avem } t(m, n) \leq c * f(m, n)\}.$$

Similar se obțin și celelalte generalizări.

## 9.3 Tehnici de analiza algoritmilor

Nu există o metodă standard pentru analiza eficienței unui algoritm. Este mai curând o chestiune de raționament, intuiție și experiență. Vom arăta pe bază de exemple cum se poate efectua o astfel de analiză.

### 9.3.1 Sortarea prin selecție

Considerăm algoritmul de sortare prin selecția minimului, reprodus mai jos:

```

pentru  $i = 1, n - 1$ 
    //se calculează poziția minimului lui  $a(i), a(i + 1), \dots, a(n)$ 
     $PozMin \leftarrow i$  //initializăm minimul cu indicele primului element
    pentru  $j = i + 1, n$ 
        dacă  $a(j) < a(PozMin)$  atunci
             $PozMin = j$ 
        sfârșit dacă
    sfârșit pentru //după  $j$ 
    //se așează minimul pe poziția  $i$ 
     $aux \leftarrow a(i)$ 
     $a(i) \leftarrow a(PozMin)$ 
     $a(PozMin) \leftarrow aux$ 
sfârșit pentru //după  $i$ 

```

Timpul necesar pentru o singură execuție a ciclului *pentru* după variabila  $j$  poate fi mărginit superior de o constantă  $a$ . În total, pentru un  $i$  fixat, ținând cont de faptul că se realizează  $n-i$  iterații, acest ciclu necesită un timp de cel mult  $b + a(n - i)$  unități, unde  $b$  este o constantă reprezentând timpul necesar pentru inițializarea buclei. O singură execuție a buclei exterioare are loc în cel mult  $c + b + a(n - i)$  unități de timp, unde  $c$  este o altă constantă. Ținând cont de faptul că bucla după  $j$  se realizează de  $n-1$  ori, timpul total de execuție al algoritmului este cel mult:

$$d + \sum_{i=1}^{n-1} (c + b + a(n - i))$$

unități de timp,  $d$  fiind din nou o constantă. Simplificăm această expresie și obținem  $\frac{a}{2}n^2 + (b + c - \frac{a}{2})n + (d - c - b)$ , de unde deducem că algoritmul necesită un timp în  $O(n^2)$ . O analiză similară asupra limitei inferioare arată că timpul este de fapt în  $\Theta(n^2)$ . Nu este necesar să considerăm cazul cel mai nefavorabil sau cazul mediu deoarece timpul de execuție al sortării prin selecție este independent de ordonarea prealabilă a elementelor de sortat.

În acest prim exemplu am analizat toate detaliile. De obicei însă, detalii cum ar fi timpul necesar inițializării ciclurilor nu se vor considera explicit, deoarece ele nu afectează ordinul de complexitate al algoritmului. Pentru cele mai multe situații, este suficient să alegem o anumită instrucțiune din algoritm ca *barometru* și să numărăm de câte ori se execută această instrucțiune. În cazul nostru, putem alege ca barometru testul

$$a[j] < a[PozMin]$$

din bucla interioară. Este ușor de observat că acest test se execută de  $\frac{n(n-1)}{2}$  ori.

### 9.3.2 Sortarea prin inserție

Timpul pentru *algoritmul de sortare prin inserție* este dependent de ordonarea prealabilă a elementelor de sortat. Algoritmul este implementat în cadrul primului volum, la capitolul *Moștenire*, secțiunea *Extinderea clasei Shape*. Analiza algoritmului se realizează pe baza implementării prezentate în cadrul acelei secțiuni. Vom folosi comparația

```
tmp.lessThan(a[j - 1])
```

din ciclul *for* ca barometru.

Să presupunem că  $p$  este fixat și fie  $n = a.length$  lungimea șirului. Cel mai nefavorabil caz apare atunci când  $tmp < a[j - 1]$  pentru fiecare  $j$  între  $p$  și 1, algoritmul făcând în această situație  $p$  comparații. Acest lucru se întâmplă (pentru fiecare valoare a lui  $p$  de la 1 la  $n - 1$ ) atunci când tabloul  $a$  este inițial ordonat descrescător. Numărul total de comparații pentru cazul cel mai nefavorabil este:

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Vom estima acum timpul mediu necesar pentru un caz oarecare. Presupunem că elementele tabloului  $a$  sunt distincte și că orice permutare a lor are aceeași probabilitate de apariție. Atunci, dacă  $1 \leq k \leq p$ , probabilitatea ca  $a[p]$  să fie cel de-al  $k$ -lea cel mai mare element dintre elementele  $a[1], a[2], \dots, a[p]$  este  $\frac{1}{p}$ . Pentru un  $p$  fixat, condiția  $a[p] < a[p - 1]$  este falsă cu probabilitatea  $\frac{1}{p}$ , deci probabilitatea ca să se execute comparația " $tmp < a[j - 1]$ " o singură dată înainte de ieșirea din bucla *while* este  $\frac{1}{p}$ . Comparația " $tmp < a[j - 1]$ " se execută de exact două ori tot cu probabilitatea  $\frac{1}{p}$  etc. Probabilitatea ca să se execute comparația de exact  $p - 1$  ori este  $\frac{2}{p}$ , deoarece aceasta se întâmplă atât când  $tmp < a[0]$  cât și când  $a[0] \leq tmp < a[1]$ ! Numărul mediu de comparații, pentru un  $p$  fixat, este în consecință, suma numărului de comparații pentru fiecare situație, înmulțită cu probabilitatea de apariție a acelei situații:

$$c_i = 1\frac{1}{i} + 2\frac{1}{i} + \dots + (i - 2)\frac{1}{i} + (i - 1)\frac{2}{i} = \frac{i+1}{2} - \frac{1}{i}$$

Pentru a sorta  $n$  elemente avem nevoie de  $\sum_{i=2}^n c_i$  comparații, ceea ce este egal cu  $\frac{n^2+3n}{4} - H_n \in \Theta(n^2)$ . Prin  $H_n = \sum_{i=1}^n i^{-1} \in \Theta(\log n)$  am notat al  $n$ -lea termen al seriei armonice.

Se observă că algoritmul de sortare prin inserare efectuează pentru cazul mediu de două ori mai puține comparații decât pentru cazul cel mai nefavorabil. Totuși, în ambele situații, numărul comparațiilor este în  $\Theta(n^2)$ .

Cu toate că algoritmul necesită un timp în  $\Omega(n^2)$  atât pentru cazul mediu cât și pentru cel mai nefavorabil caz, pentru cazul cel mai favorabil (când inițial tabloul este ordonat crescător) timpul este în  $O(n)$ . De fapt, pentru cazul cel mai favorabil, timpul este și în  $\Omega(n)$  (deci în  $\Theta(n)$ ).

### 9.3.3 Turnurile din Hanoi

Matematicianul francez Eduard Lucas a propus în 1883 o problemă care a devenit apoi celebră mai ales datorită faptului că a prezentat-o sub forma unei legende. Se spune că Brahma (Zeul Creației la hinduși) a fixat pe Pământ trei tije de diamant și pe una din ele a pus în ordine crescătoare 64 de discuri de aur de dimensiuni diferite, astfel încât discul cel mai mare era jos. Brahma a creat și o mănăstire, iar sarcina călugărilor era să mute toate discurile pe o altă tijă. Singura operațiune permisă era mutarea câte unui singur disc de pe o tijă pe alta, astfel încât niciodată să nu se pună un disc mai mare peste un disc mai mic. Legenda spune că sfârșitul lumii se va petrece atunci când călugării vor săvârși lucrarea. Vom vedea că aceasta se dovedește a fi o previziune extrem de optimistă asupra sfârșitului lumii. Presupunând că în fiecare secundă se mută un disc și se lucrează fără întrerupere, cele 64 de discuri nu pot fi mutate nici în 500 de miliarde de ani de la începutul acțiunii!

Pentru a rezolva problema, vom numerota cele trei tije cu 1, 2 și respectiv 3. Se observă că pentru a muta cele  $n$  discuri de pe tija cu numărul  $i$  pe tija cu numărul  $j$  ( $i$  și  $j$  iau valori între 1 și 3) este necesar să transferăm primele  $n - 1$  discuri de pe tija  $i$  pe tija  $6 - i - j$  (adică pe tija rămasă liberă), apoi să transferăm discul  $n$  de pe tija  $i$  pe tija  $j$ , iar apoi retransferăm cele  $n - 1$  discuri de pe tija  $6 - i - j$  pe tija  $j$ . Cu alte cuvinte, reducem problema mutării a  $n$  discuri la problema mutării a  $n - 1$  discuri. Următoarea metodă Java descrie acest algoritm recursiv.

Listing 9.1: Metoda hanoi

```
1 public static void hanoi(int n, int i, int j)
2 {
3     if (n > 0)
4     {
5         hanoi(n - 1, i, 6 - i - j);
6         System.out.println(i + "—>" + j);
7         hanoi(n - 1, 6 - i - j, j);
8     }
9 }
```

Pentru rezolvarea problemei inițiale, facem apelul

```
hanoi(64, 1, 2);
```

Considerăm instrucțiunea `println` ca barometru. Timpul necesar algoritmului este exprimat prin următoare recurență:

$$t(n) = \begin{cases} 1 & \text{dacă } n = 1 \\ 2t(n-1) + 1 & \text{dacă } n > 1 \end{cases}$$

Vom demonstra că  $t(n) = 2^n - 1$ . Rezultă că  $t \in \Theta(2^n)$ .

Acest algoritm este optim în sensul că este imposibil să mutăm discuri de pe o tijă pe alta cu mai puțin de  $2^n - 1$  operații. Pentru a muta 64 de discuri vor fi în consecință necesare un număr astronomic de  $2^{64}$  operații. Implementarea în oricare limbaj de programare care admite exprimarea recursivă se poate face aproape în mod direct.

## 9.4 Analiza algoritmilor recursivi

Am văzut în exemplul precedent cât de puternică și în același timp cât de elegantă este recursivitatea în elaborarea unui algoritm. Cel mai important câștig al exprimării recursive este faptul că ea este naturală și compactă, fără să ascundă esența algoritmului prin detaliile de implementare. Pe de altă parte, apelurile recursive trebuie folosite cu discernământ, deoarece solicită și ele resursele calculatorului (timp și memorie). Analiza unui algoritm recursiv implică aproape întotdeauna rezolvarea unui sistem de recurențe. Vom vedea în continuare cum pot fi rezolvate astfel de recurențe. Începem cu tehnica cea mai simplă.

### 9.4.1 Metoda iterației

Cu puțină experiență și intuiție putem rezolva de multe ori astfel de recurențe prin *metoda iterației*: se execută primii pași, se intuiește forma generală, iar apoi se demonstrează prin inducție matematică că forma este corectă. Să considerăm de exemplu recurența problemei turnurilor din Hanoi. Se observă că pentru a muta  $n$  discuri este necesar să mutăm  $n - 1$  discuri, apoi să mutăm un disc și în final din nou  $n - 1$  discuri. În consecință, pentru un anumit  $n > 1$  obținem succesiv:

$$t(n) = 2t(n-1) + 1 = 2^2t(n-2) + 2 + 1 = \dots = 2^{n-1}t(1) + \sum_{i=0}^{n-2} 2^i$$

Rezultă că  $t(n) = 2^n - 1$ . Prin inducție matematică se demonstrează acum cu ușurință că această formă generală este corectă.

### 9.4.2 Inducția constructivă

Inducția matematică este folosită de obicei ca tehnică de demonstrare a unei aserțiuni deja enunțate. Vom vedea în această secțiune că inducția matematică poate fi utilizată cu succes și în descoperirea parțială a enunțului aserțiunii. Aplicând această tehnică, putem simultan să demonstrăm o aserțiune doar parțial specificată și să descoperim specificațiile care lipsesc și datorită cărora aserțiunea este corectă. Vom vedea că această tehnică a *inducției constructive* este utilă pentru rezolvarea anumitor recurențe care apar în contextul analizei algoritmilor. Începem cu un exemplu.

Fie funcția  $f : \mathbf{N} \rightarrow \mathbf{N}$  definită prin recurența:

$$f(n) = \begin{cases} 0 & \text{dacă } n = 1 \\ f(n-1) + n & \text{altfel} \end{cases}$$

Să presupunem pentru moment că nu știm că  $f(n) = \frac{n(n+1)}{2}$ . Avem

$$f(n) = \sum_{i=0}^n i \leq \sum_{i=0}^n n = n^2$$

și deci  $f(n) \in O(n^2)$ . Aceasta ne sugerează să formulăm *ipoteza inducției specificate parțial*  $IISP(n)$  conform căreia  $f$  este de forma  $f(n) = an^2 + bn + c$ . Această ipoteză este parțială în sensul că  $a$ ,  $b$  și  $c$  nu sunt încă cunoscute. Tehnica inducției constructive constă în a demonstra prin inducție matematică această ipoteză incompletă și a determina în același timp valorile constantelor necunoscute  $a$ ,  $b$  și  $c$ .

Presupunem că  $IISP(n-1)$  este adevărată pentru un anumit  $n \geq 1$ . Atunci,  $f(n-1) = a(n-1)^2 + b(n-1) + c = an^2 + (1+b-2a)n + (a-b+c)$ . Dacă dorim să arătăm că  $IISP(n)$  este adevărată, trebuie să arătăm că  $f(n) = an^2 + bn + c$ . Prin identificarea coeficienților puterilor lui  $n$ , obținem ecuațiile  $1+b-2a = b$  și  $a-b+c = c$ , cu soluția  $a = b = \frac{1}{2}$ ,  $c$  putând fi oarecare. Avem acum o ipoteză "mai completă" (abuzul de limbaj este inevitabil), pe care o numim tot  $IISP(n)$ ,  $f(n) = \frac{n^2}{2} + \frac{n}{2} + c$ . Am arătat că dacă  $IISP(n-1)$  este adevărată pentru un anumit  $n \geq 1$ , atunci este adevărată și  $IISP(n)$ . Rămâne să arătăm că este adevărată și  $IISP(0)$ . Trebuie să arătăm că  $f(0) = a0^2 + b0 + c$ . Știm că  $f(0) = 0$ , deci  $IISP(0)$  este adevărată pentru  $c = 0$ . În concluzie am demonstrat că  $f(n) = \frac{n^2}{2} + \frac{n}{2}$  pentru orice  $n$ .

### 9.4.3 Recurențe liniare omogene

Există din fericire și tehnici care pot fi folosite aproape automat pentru a rezolva anumite clase de recurențe. Vom începe prin a considera ecuații recurente liniare omogene, adică ecuații de forma:

$$a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = 0 \quad (*)$$

unde  $t_i$  sunt valorile pe care le căutăm, iar coeficienții  $a_i$  sunt constante.

Conform intuiției<sup>2</sup>, vom căuta soluții de forma:

$$t_n = x^n$$

unde  $x$  este o constantă (deocamdată necunoscută). Dacă înlocuim această soluție în ecuația (\*), obținem

$$a_0 x^n + a_1 x^{n-1} + \dots + a_k x^{n-k} = 0$$

Soluțiile acestei ecuații sunt fie soluția trivială  $x = 0$ , care nu ne interesează, fie soluțiile ecuației:

$$a_0 x^k + a_1 x^{k-1} + \dots + a_k = 0$$

care se numește *ecuația caracteristică* a recurenței liniare și omogene (\*).

Presupunând deocamdată că cele  $k$  rădăcini  $r_1, r_2, \dots, r_k$  ale acestei ecuații caracteristice sunt distincte, se verifică ușor că orice combinație liniară

$$t_n = \sum_{i=1}^k c_i r_i^n$$

este o soluție a recurenței (\*), unde constantele  $c_1, c_2, \dots, c_k$  sunt determinate de condițiile inițiale. Se poate demonstra faptul că (\*) are soluții *numai* de această formă.

Să exemplificăm prin recurența care definește șirul lui Fibonacci

$$t_n = t_{n-1} + t_{n-2}, n \geq 2$$

iar  $t_0 = 0, t_1 = 1$ . Putem să rescriem această recurență sub forma

$$t_n - t_{n-1} - t_{n-2} = 0$$

care are ecuația caracteristică

$$x^2 - x - 1 = 0$$

cu rădăcinile  $r_{1,2} = \frac{1 \pm \sqrt{5}}{2}$ . Soluția generală are forma:

$$t_n = c_1 r_1^n + c_2 r_2^n$$

<sup>2</sup>De fapt, adevărul este că aici nu este vorba de intuiție, ci de experiență.

Impunând condițiile inițiale,  $t_0 = 0, t_1 = 1$ , obținem

$$c_1 + c_2 = 0, n = 0$$

$$c_1 r_1 + c_2 r_2 = 1, n = 1$$

de unde determinăm

$$c_{1,2} = \mp \frac{1}{\sqrt{5}}$$

Deci,  $t_n = \frac{1}{\sqrt{5}}(r_1^n - r_2^n)$ . Observăm că  $r_1 = \phi, r_2 = -\phi^{-1}$  și obținem:

$$t_n = \frac{1}{\sqrt{5}}(\phi^n - (-\phi)^{-n})$$

care este cunoscuta relație a lui Moivre, descoperită la începutul secolului XVIII. Nu prezintă nici o dificultate să arătăm acum că timpul pentru calculul recursiv al șirului lui Fibonacci este în  $\Theta(\phi^n)$ .

Cum procedăm însă atunci când rădăcinile ecuației caracteristice nu sunt distincte? Se poate arăta că dacă  $r$  este o rădăcină de multiplicitate  $m$  a ecuației caracteristice, atunci  $t_n = r^n, t_n = nr^n, t_n = n^2 r^n, \dots, t_n = n^{m-1} r^n$  sunt soluții pentru (\*). Soluția generală pentru o astfel de recurență este atunci o combinație liniară a acestor termeni și a termenilor proveniți de la celelalte rădăcini ale ecuației caracteristice. Din nou, sunt de determinat exact  $k$  constante din condițiile inițiale.

Vom da din nou un exemplu. Fie recurența

$$t_n = 5t_{n-1} - 8t_{n-2} + 4t_{n-3}$$

cu  $t_0 = 0, t_1 = 1, t_2 = 2$ . Ecuația caracteristică are rădăcinile 1 (de multiplicitate 1) și 2 (de multiplicitate 2). Soluția generală este:

$$t_n = c_1 1^n + c_2 2^n + c_3 n 2^n$$

Din condițiile inițiale, obținem  $c_1 = -2, c_2 = 2, c_3 = -\frac{1}{2}$ .

#### 9.4.4 Recurențe liniare neomogene

Considerăm acum recurențe de următoarea formă mai generală

$$a_0 t_n + a_1 t_{n-1} + \dots + a_k t_{n-k} = b^n p(n) \quad (**)$$



unde  $b$  este o constantă, iar  $p(n)$  este un polinom în  $n$  de grad  $d$ . Ideea generală este ca prin manipulări convenabile să reducem un astfel de caz la o formă omogenă.

De exemplu, o astfel de recurență poate fi:

$$t_n - 2t_{n-1} = 3^n$$

În acest caz  $b = 3$  și  $p(n) = 1$  un polinom de grad 0. O simplă manipulare ne permite să reducem acest exemplu la forma  $(*)$ . Înmulțind recurența cu 3, obținem:

$$3t_n - 6t_{n-1} = 3^{n+1}$$

Înlocuind pe  $n$  cu  $n + 1$  în recurența originală, avem:

$$t_{n+1} - 2t_n = 3^{n+1}$$

În final, scădem aceste două ecuații și obținem:

$$t_{n+1} - 5t_n + 6t_{n-1} = 0$$

Am obținut o recurență omogenă pe care o putem rezolva ca în secțiunea precedentă. Ecuația caracteristică este:

$$x^2 - 5x + 6 = 0$$

adică  $(x - 2)(x - 3) = 0$ .

Intuitiv, observăm că factorul  $(x - 2)$  corespunde părții stângi a recurenței originale, în timp ce factorul  $(x - 3)$  a apărut ca rezultat al manipuleșilor efectuate pentru a scăpa de partea dreaptă.

Iată un al doilea exemplu:

$$t_n - 2t_{n-1} = (n + 5)3^n$$

Manipulările necesare sunt puțin mai complicate. Trebuie să:

1. Înmulțim recurența cu 9;
2. Înlocuim în recurență pe  $n$  cu  $n + 2$ ;
3. Înlocuim în recurență pe  $n$  cu  $n + 1$  și să înmulțim apoi cu -6.

Adunând cele trei ecuații obținute anterior avem:

$$t_{n+2} - 8t_{n+1} + 21t_n - 18t_{n-1} = 0$$

Am ajuns din nou la o ecuație omogenă. Ecuația caracteristică corespunzătoare este

$$x^3 - 8x^2 + 21x - 18 = 0$$

adică  $(x - 2)(x - 3)^2$ .

Încă o dată, observăm că factorul  $(x - 2)$  provine din partea stângă a recurenței originale, în timp ce factorul  $(x - 3)^2$  este rezultatul manipulării.

Generalizând acest procedeu, se poate arăta că pentru a rezolva  $(**)$  este suficient să luăm următoarea ecuație caracteristică:

$$(a_0x^k + a_1x^{k-1} + \dots + a_k)(x - b)^{d+1} = 0$$

Odată ce s-a obținut această ecuație, se procedează ca în cazul omogen.

Vom rezolva acum recurența corespunzătoare problemei turnurilor din Hanoi

$$t_n = 2t_{n-1} + 1, n \geq 1$$

iar  $t_0 = 0$ . Rescriem recurența astfel

$$t_n - 2t_{n-1} = 1$$

care este de forma  $(**)$  cu  $b = 1$  și  $p(n) = 1$ , un polinom cu grad 0. Ecuația caracteristică este atunci  $(x - 1)(x - 2)$ , cu soluțiile 1 și 2. Soluția generală a recurenței este:

$$t_n = c_1 1^n + c_2 2^n$$

Avem nevoie de două condiții inițiale. Știm că  $t_0 = 0$ ; pentru a găsi cea de-a doua condiție calculăm

$$t_1 = 2t_0 + 1$$

Din condițiile inițiale, obținem  $t_n = 2^n - 1$ .

*Observație:* dacă ne interesează doar ordinul lui  $t_n$ , nu este necesar să calculăm efectiv constantele în soluția generală. Dacă știm că  $t_n = c_1 1^n + c_2 2^n$ , rezultă că  $t_n \in O(2^n)$ . Din faptul că numărul de mutări a unor discuri nu poate fi negativ sau constant (deoarece avem în mod evident  $t_n \geq n$ ), deducem că  $c_2 > 0$ . Avem atunci  $t_n \in \Omega(2^n)$  și deci  $t_n \in \Theta(2^n)$ . Putem obține chiar ceva mai mult.

Substituind soluția generală înapoi în recurența originală, găsim

$$1 = t_n - 2t_{n-1} = c_1 + c_2 2^n - 2(c_1 + c_2 2^{n-1}) = -c_1$$

Indiferent de condiția inițială,  $c_1$  este -1.

### 9.4.5 Schimbarea variabilei

Uneori putem rezolva recurențe mai complicate printr-o schimbare de variabilă. În exemplele care urmează, vom nota cu  $T(n)$  termenul general al recurenței și cu  $t_k$  termenul noii recurențe obținute printr-o schimbare de variabilă. Presupunem pentru început că  $n$  este o putere a lui 2.

Un prim exemplu este recurența

$$T(n) = 4T\left(\frac{n}{2}\right) + n, n > 1$$

în care înlocuim pe  $n$  cu  $2^k$ , notăm  $t_k = T(2^k) = T(n)$  și obținem:

$$t_k = 4t_{k-1} + 2^k$$

Ecuția caracteristică a acestei recurențe liniare este (conform paragrafului anterior):

$$(x - 4)(x - 2) = 0$$

și deci  $t_k = c_1 4^k + c_2 2^k$ . Înlocuim pe  $k$  cu  $\lg n$

$$T(n) = c_1 n^2 + c_2 n$$

Rezultă că  $T(n) \in O(n^2 \mid n \text{ este o putere a lui } 2)$ .

Un al doilea exemplu îl reprezintă ecuația

$$T(n) = 4T\left(\frac{n}{2}\right) + n^2, n > 1$$

Procedând la fel, ajungem la recurența

$$t_k = 4t_{k-1} + 4^k$$

cu ecuația caracteristică

$$(x - 4)^2 = 0$$

și soluția generală  $t_k = c_1 4^k + c_2 k 4^k$ . Atunci,

$$T(n) = c_1 n^2 + c_2 n^2 \lg n$$

și obținem că  $T(n) \in O(n^2 \log n \mid n \text{ este o putere a lui } 2)$ .

În fine, să considerăm și exemplul

$$T(n) = 3T\left(\frac{n}{2}\right) + cn, n > 1$$

$c$  fiind o constantă. Obținem succesiv

$$T(2^k) = 3T(2^{k-1}) + c2^k$$

$$t_k = 3t_{k-1} + c2^k$$

cu ecuația caracteristică

$$(x - 3)(x - 2) = 0$$

$$t_k = c_1 3^k + c_2 2^k$$

$$T(n) = c_1 3^{\lg n} + c_2 n$$

și, deoarece

$$a^{\lg b} = b^{\lg a}$$

obținem

$$T(n) = c_1 n^{\lg 3} + c_2 n$$

deci,  $T(n) \in O(n^{\lg 3} \mid n \text{ este o putere a lui } 2)$ .

În toate aceste exemple am folosit notația asimptotică condiționată. Pentru a arăta că rezultatele obținute sunt adevărate pentru orice  $n$ , este suficient să adăugăm condiția ca  $T(n)$  să fie crescătoare pentru  $n > n_0$ .

Putem enunța acum o proprietate care este utilă ca rețetă pentru analiza algoritmilor cu recursivități de forma celor din exemplele precedente. Proprietatea, a cărei demonstrare o lăsăm ca exercițiu, este foarte utilă la analiza algoritmilor Divide et Impera.

*Propoziție.* Fie  $T : \mathbb{N} \rightarrow \mathbb{R}^+$  o funcție nedescrescătoare

$$T(n) = aT\left(\frac{n}{b}\right) + cn^k, n > n_0$$

unde  $n_0 \geq 1$ ,  $b \geq 2$  și  $k \geq 0$  sunt întregi,  $a$  și  $c$  sunt numere reale pozitive, iar  $\frac{n}{n_0}$  este o putere a lui  $b$ . Atunci avem:

$$T(n) \in \begin{cases} \Theta(n^k) & \text{dacă } a < b^k \\ \Theta(n^k \log n) & \text{dacă } a = b^k \\ \Theta(n^{\log_b a}) & \text{dacă } a > b^k \end{cases}$$

## 9.5 Implementarea algoritmilor

Am considerat utilă prezența la sfârșitul capitolului a implementărilor Java pentru problemele prezentate pe parcursul acestui capitol. Este cazul algoritmilor de sortare prin selecție, a celui de sortare prin inserție și a algoritmului turnurilor din Hanoi.

Listing 9.2: Implementarea algoritmului de sortare prin selecția minimului

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Sortare prin selectia minimului.
6  */
7 public class SortareSelMin
8 {
9
10  /** Ordonarea prin selectia minimului. */
11  public static void ordonare(int [] a)
12  {
13      for (int i = 0; i < a.length - 1; i++)
14      {
15          int pozMin = i;
16
17          for (int j = i + 1; j < a.length; j++)
18          {
19              if (a[j] < a[pozMin])
20              {
21                  pozMin = j;
22              }
23          }
24
25          int aux = a[i];
26          a[i] = a[pozMin];
27          a[pozMin] = aux;
28      }
29  }
30
31  /** Programul principal. */
32  public static void main(String [] args)
33  {
34      //citirea elementelor sirului
35      System.out.println("Introduceti elementele sirului " +
36          " (pe aceeasi linie, separate prin spatiu):");
37      int [] s = Reader.readIntArray();
38
39      //ordonarea sirului s
40      ordonare(s);

```

```
41
42     //afisare rezultate
43     System.out.print("Sirul ordonat este: ");
44     for (int i = 0; i < s.length; i++)
45     {
46         System.out.print(s[i] + " ");
47     }
48     System.out.println();
49 }
50 }
```

---

Listing 9.3: Implementarea algoritmului de sortare prin inserție

---

```
1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Sortare prin insertie.
6  */
7 public class SortareInsertie
8 {
9
10     /** Ordonarea prin insertie.*/
11     public static void ordonare(int[] a)
12     {
13         for (int i = 1; i < a.length; i++)
14         {
15             int tmp = a[i];
16             int j = i;
17
18             for (; (j > 0) && (tmp < a[j - 1]); j--)
19             {
20                 a[j] = a[j - 1];
21             }
22
23             a[j] = tmp;
24         }
25     }
26
27     /** Programul principal.*/
28     public static void main(String[] args)
29     {
30         //citirea elementelor sirului
31         System.out.println("Introduceti elementele sirului" +
32             " (pe aceeasi linie, separate prin spatiu):");
33         int[] s = Reader.readIntArray();
34
35         //ordonarea sirului s
36         ordonare(s);
37     }
38 }
```

```

38 //afisare rezultate
39 System.out.print("Sirul ordonat este: ");
40 for (int i = 0; i < s.length; i++)
41 {
42     System.out.print(s[i] + " ");
43 }
44 System.out.println();
45 }
46 }

```

Listing 9.4: Implementarea problemei turnurilor din Hanoi

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Turnurile din Hanoi.
6  */
7 public class Hanoi
8 {
9     /** Implementarea algoritmului "Turnurile din Hanoi". */
10    public static void hanoi(int n, int i, int j)
11    {
12        if (n > 0)
13        {
14            hanoi(n - 1, i, 6 - i - j);
15            System.out.println(i + "--> " + j);
16            hanoi(n - 1, 6 - i - j, j);
17        }
18    }
19
20    /** Programul principal. */
21    public static void main(String[] args)
22    {
23        //citirea numarului de discuri
24        System.out.print("Introduceti numarul de discuri: ");
25
26        int n = Reader.readInt();
27
28        //apelul metodei hanoi
29        hanoi(n, 1, 2);
30    }
31 }

```

## Rezumat

Capitolul de față a realizat o scurtă introducere în domeniul vast al analizei algoritmilor. Ideea cea mai importantă care reiese de aici este că algoritmii

utilizați afectează timpul de execuție al unui program mult mai drastic decât artificiile de programare. Algoritmii care au un timp de lucru exponențial nu sunt în general aplicabili pentru date de intrare de dimensiuni rezonabile, spre deosebire de cei liniari sau pătratici al căror timp de execuție nu crește atât de drastic odată cu dimensiunea datelor de intrare.

Capitolul următor prezintă cele mai importante structuri de date, împreună cu gradul lor de eficiență și cu operațiile pe care le permit. De asemenea, sunt prezentate diverse situații în care pot fi folosite structurile de date. Restul capitolelor sunt dedicate prezentării algoritmilor fundamentali.

## Noțiuni fundamentale

$O(f)$ : notație utilizată pentru a determina termenul dominant al funcției  $f$ . Cu ajutorul ei se limitează superior timpul de execuție al unui algoritm.

$\Omega(f)$ : notație utilizată pentru a limita inferior timpul de execuție al unui algoritm.

$\Theta(f)$ : notație utilizată pentru a arăta că timpul de execuție al unui algoritm este limitat atât inferior cât și superior de câte un multiplu real al aceleiași funcții  $f$ .

**algoritm liniar**: algoritm care are timpul de execuție  $O(n)$ .

**algoritm exponențial**: algoritm al cărui timp de execuție crește exponențial în raport cu dimensiunea datelor de intrare ( $O(a^n)$ ).

**algoritm polinomial**: algoritm al cărui timp de execuție este mărginit superior de un polinom ( $O(n^k)$ ).

**ecuație caracteristică**: ecuație polinomială atașată recurențelor liniare, prin a cărei rezolvare se găsește soluția recurenței.

**inducția constructivă**: o variantă a inducției matematice care permite demonstrarea unei aserțiuni parțial sau incomplet enunțate precum și descoperirea specificațiilor care lipsesc din aserțiunea respectivă.

**metoda iterației**: metodă simplă de rezolvare a recurențelor liniare în care se execută primii pași ai recurenței după care se intuiește forma generală care se demonstrează prin inducție.

**recurență liniară**: formulă de recurență în care termenul al  $n$ -lea este exprimat ca o combinație liniară a termenilor precedenți.

**schimbarea variabilei**: tehnică utilizată pentru a reduce anumite formule de recurență la o formă liniară.



## Erori frecvente

1. Pentru cicluri iterative imbricate, timpul total de execuție este produsul dimensiunilor ciclurilor, în timp ce în cazul ciclurilor iterative consecutive, afirmația nu este adevărată. De exemplu, pentru două cicluri imbricate care se execută fiecare de la 1 la  $n^2$ , timpul total de execuție este  $O(n^4)$ .
2. Nu scrieți expresii de tipul  $O(2n^2)$  sau  $O(n^2 + n)$ . În cele mai multe situații, este necesar doar termenul dominant, fără constanta care îl precede. În consecință notația utilizată pentru ambele cazuri de mai înainte este  $O(n^2)$ .
3. Pentru a exprima limita inferioară a complexității unui algoritm, utilizați notația  $\Omega$ , nu  $O$ .
4. Baza în care se scrie logaritmul este irelevantă pentru notațiile asimptotice. Astfel,  $O(\lg n)$  este egal cu  $O(\ln n)$  deoarece ele diferă doar printr-o constantă multiplicativă.

## Exerciții

1. Care din următoarele afirmații sunt adevărate?
  - (a)  $n^2 \in O(n^3)$
  - (b)  $n^3 \in O(n^2)$
  - (c)  $2^{n+1} \in O(2^n)$
  - (d)  $(n+1)! \in O(n!)$
  - (e) pentru orice funcție  $f : \mathbf{N} \rightarrow \mathbf{R}^*$ ,  $f \in O(n) \Rightarrow [f^2 \in O(n^2)]$
  - (f) pentru orice funcție  $f : \mathbf{N} \rightarrow \mathbf{R}^*$ ,  $f \in O(n) \Rightarrow [2^f \in O(2^n)]$
2. Demonstrați că relația " $\in O$ " este *tranzitivă*: dacă  $f \in O(g)$  și  $g \in O(h)$ , atunci  $f \in O(h)$ . Deduceți de aici că dacă  $g \in O(h)$ , atunci  $O(g) \subseteq O(h)$ .
3. Găsiți două funcții  $f, g : \mathbf{N} \rightarrow \mathbf{R}^*$ , astfel încât  $f \notin O(g)$  și  $g \notin O(f)$ .

*Soluție:*  $f(n) = n, g(n) = n^{1+\sin n}$ .

4. Pentru oricare două funcții  $f, g : \mathbf{N} \rightarrow \mathbf{R}^*$  definim următoarea relație binară:  $f \leq g$  dacă  $O(f) \subseteq O(g)$ . Demonstrați că relația " $\leq$ " este o relație de ordine parțială în mulțimea funcțiilor definite pe  $\mathbf{N}$  și cu valori în  $\mathbf{R}^*$ .

*Indicație: Trebuie arătat că relația este reflexivă, tranzitivă și antisimetrică. Țineți cont de exercițiul 3.*

5. Pentru oricare două funcții  $f, g : \mathbf{N} \rightarrow \mathbf{R}^*$  demonstrați că  $O(f + g) = O(\max(f, g))$  unde suma și maximul se iau punctual.
6. Fie  $f(n) = a_m n^m + \dots + a_1 n + a_0$  un polinom de grad  $m$ , cu  $a_m > 0$ . Arătați că  $f \in O(n^m)$ .
7. Considerăm afirmația  $O(n^2) = O(n^3 + (n^2 - n^3)) = O(\max(n^3, n^2 - n^3)) = O(n^3)$ . Unde este eroarea?
8. Considerăm afirmația  $\sum_{i=1}^n i = 1 + 2 + \dots + n \in O(1 + 2 + \dots + n) = O(\max(1 + 2 + \dots + n)) = O(n)$ . Unde este eroarea?
9. Pentru oricare două funcții  $f, g : \mathbf{N} \rightarrow \mathbf{R}^*$  demonstrați că  $\Theta(f) + \Theta(g) = \Theta(f + g) = \Theta(\max(f, g)) = \max(\Theta(f), \Theta(g))$ , unde suma și maximul se iau punctual.

10. Analizați eficiența următorilor algoritmi:

- (a) pentru  $i=1, n$   
       pentru  $j=1, 5$   
           {operație elementară}
- (b) pentru  $i=1, n$   
       pentru  $j=1, i+1$   
           {operație elementară}
- (c) pentru  $i=1, n$   
       pentru  $j=1, 6$   
           pentru  $k=1, n$   
               {operație elementară}
- (d) pentru  $i=1, n$   
       pentru  $j=1, i$   
           pentru  $k=1, n$   
               {operație elementară}

11. Construiți un algoritm cu timpul în  $\Theta(n \log n)$ .

12. Fie un algoritm:

pentru  $i=0,n$

$j \leftarrow i$

cât timp  $j < > 0$

$j \leftarrow j \div 2$

Găsiți ordinul exact al timpului de execuție.

13. Rezolvați următoarea recurență:  $t_n - 3t_{n-1} - 4t_{n-2} = 0, n \geq 2$  cu  $t_0 = 0, t_1 = 1$ .

14. Care este timpul de execuție pentru un algoritm recursiv cu recurența:  $t_n = 2t_{n-1} + n$ ?

*Indicație: Se ajunge la ecuația caracteristică  $(x - 2)(x - 1)^2 = 0$ , iar soluția generală este  $t_n = c_1 2^n + c_2 1^n + c_3 n 1^n$ . Rezultă că  $t_n \in O(2^n)$ . Substituind soluția generală în recurență, obținem că, indiferent de condiția inițială,  $c_2 = -2$  și  $c_3 = -1$ . Atunci, toate soluțiile interesante ale recurenței trebuie să aibă  $c_1 > 0$  și ele sunt toate în  $\Omega(2^n)$ , deci în  $\Theta(2^n)$ .*

15. Să se calculeze secvența de sumă maximă, formată din termeni consecutivi, ai unui șir de numere.

## 10. Structuri de date

Nu poți obține întotdeauna ceea ce dorești, dar, dacă încerci, uneori vei obține ceea ce ai nevoie.

---

Autor anonim

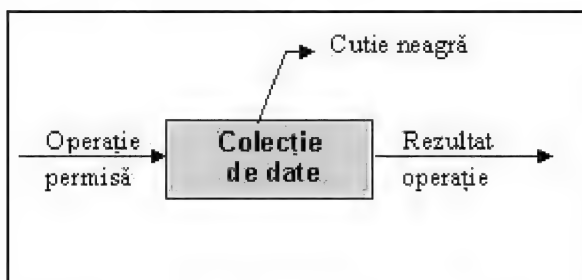
Două programe care rezolvă aceeași problemă pot să arate complet diferit. Unul poate fi extrem de lizibil, concis și ușor de modificat pentru a se adapta la rezolvarea unor probleme asemănătoare, iar celălalt poate fi impenetrabil, obscur, interminabil și dificil de modificat. Cele două programe pot să difere atât de mult în ceea ce privește durata de execuție și necesarul de memorie încât, pentru un anumit set de intrare, unul furnizează răspunsul după 2 secunde, iar celălalt după câteva secole!

Experiența a demonstrat că aceste diferențe sunt generate de structura programului și de structura datelor.

Programele sunt scrise pentru a rezolva probleme reale. Structurarea programului împarte problema și soluția ei în componente mai simple și mai ușor de înțeles. Informația care trebuie procesată este reținută în structuri de date (tablouri, înregistrări, liste, stive, arbori, fișiere etc.). O structură de date grupează datele. O structură de date aleasă adecvat poate face operațiile simple și eficiente, iar una aleasă neadecvat poate face operațiile alambicate și ineficiente.

Structurile de date conțin informație asupra căreia se operează în timpul execuției unui program. Despre programe obișnuim să spunem că procesează informație, când, de fapt, ele procesează structuri de date. Astfel, nu mai pare surprinzător faptul că structura datelor și a programelor sunt extrem de importante și că ele trebuie corelate corespunzător pentru a programa cu succes. Asimilarea cunoștințelor legate de structuri de date și algoritmi, alături de stăpânirea conceptelor fundamentale ale programării orientate pe obiecte, îi permit programatorului să mențină o supremație regală asupra programelor, astfel încât

Figura 10.1: Închiderea datelor într-o cutie neagră. Datele pot fi accesate doar prin invocarea unei operații permise.



acestea să rămână lizibile, ușor de întreținut și eficiente, chiar și atunci când cresc în dimensiune.

Atât programele cât și datele au o anumită structurare și atât structura programului cât și a datelor trebuie să fie adecvată problemei de rezolvat. Vom urmări să prezentăm structurile de date nu ca un subiect teoretic izolat, ci ca pe un instrument esențial al procesului de rezolvare a problemelor care conduce la crearea de programe eficiente.

Structurile de date sunt importante, deoarece modul în care programatorul alege să reprezinte datele afectează în mod semnificativ claritatea, concizia, viteza de execuție și necesarul de memorie al programului. În acest capitol, cât și în cele care urmează, vom prezenta cum să folosim diferite structuri de date pentru a crea programe corecte și eficiente. Vom vedea că operațiile care trebuie realizate asupra datelor sunt cele care determină care este cea mai potrivită structură de date care trebuie folosită.

Dezvoltarea de programe este dificilă, mai ales atunci când este făcută fără nici un fel de strategie. Programarea orientată pe obiecte, prezentată în prima parte a lucrării, ușurează mult dezvoltarea programelor și conduce la programe bine structurate. Ea constă în împărțirea atentă a unei probleme complexe în componente mai simple a căror interfață este apoi cu ușurință combinată pentru a rezolva problema inițială.

Abstractizarea datelor constă în a trata o colecție de date extrăgând aspectele sale esențiale, ignorând pe cât se poate detaliile. Abstractizarea datelor reduce datele la o colecție și la operațiile care se pot realiza asupra acestei colecții. Efectul este ca și cum colecția de date ar fi închisă într-o cutie neagră (black box) impenetrabilă, singurul mod de a accesa datele fiind invocarea uneia sau mai multor operații permise (**Figura 10.1**).

---

Modul în care datele sunt așezate în acea cutie neagră și modul în care operațiile se execută devin detalii irelevante<sup>1</sup>. Astfel de detalii determină eficiența programului, dar nu îi afectează structura logică.

Abstractizarea datelor trebuie privită mai degrabă ca o facilitare care ușurează efortul de programare, și nu ca o nouă constrângere în privința stilului de programare. Pentru a înțelege mai bine aceste noțiuni, să vedem cum poate fi definită simplificat o stivă ca tip abstract de date. Stiva este o colecție de date omogene (de același tip) asupra căreia se pot realiza următoarele operații:

- PUSH(X) - are ca efect depunerea lui X pe vârful stivei;
- POP(X) - are ca efect încărcarea valorii din vârful stivei în parametrul X și eliminarea vârfului stivei.

Modul în care cele două operații (PUSH și POP) sunt implementate și modul în care datele sunt reținute în stivă (static, înlănuțit etc.) nu trebuie să transpară utilizatorului; pe el pur și simplu nu trebuie să îl intereseze acest lucru. Este exact ca și curentul electric: atunci când acționăm comutatorul de la veioză, știm că becul se va aprinde; care este procesul prin care filamentul becului se încinge și emite lumină nu ne privește. Tot astfel, și listele, cozile, arborii binari împreună cu operațiile care se fac asupra lor pot fi privite ca tipuri abstracte de date.

Unui tip abstract de date *mulțime* putem să-i asociem operații cum ar fi reuniune, intersecție, dimensiune, complementară. Într-o altă situație, putem avea nevoie numai de operatorii de reuniune și apartenență, care definesc un alt tip abstract de date (TAD), care poate să aibă o cu totul altă organizare internă, deoarece, așa cum am spus, operațiile sunt cele care definesc TAD-ul.

În consecință, rolul acestui capitol este de a prezenta modul în care se construiesc aceste tipuri abstracte de date (sau structuri de date) pentru a le putea utiliza apoi în crearea de programe robuste, lizibile și eficiente. Ideea de bază este că implementarea operațiilor unui TAD se realizează o singură dată în cadrul unei aplicații, și orice altă parte a aplicației va utiliza structura de date prin intermediul interfeței pe care aceasta o expune.

Dacă din diverse motive anumite detalii de implementare trebuie schimbate, acest lucru se va face ușor prin modificarea rutinelor care realizează operațiile din TAD. Aceste schimbări nu vor afecta în nici un fel restul programului, deoarece interfața se menține neschimbată. În exemplul nostru cu stiva, dacă decidem să trecem de la alocarea secvențială (statică) a elementelor stivei la

---

<sup>1</sup>Irelevante pentru cel care utilizează structura de date. Noi ne vom ocupa în acest capitol tocmai de modul de construcție al acestei cutii negre, pentru a o putea utiliza apoi ori de câte ori este necesar.

alocarea înlănțuită (dinamică), implementarea operațiilor PUSH și POP va trebui în mod cert schimbată; această schimbare nu va afecta în nici un fel restul programului, care va utiliza operațiile PUSH și POP expuse de stivă fără a sesiza că implementarea acestora este diferită.

Acest capitol prezintă șase dintre cele mai cunoscute structuri de date: stive, cozi, liste înlănțuite, arbori binari de căutare, tabele de repartizare (engl. hash-tables) și cozi de prioritate. Scopul este acela de a defini fiecare structură de date și de a furniza o estimare intuitivă pentru complexitatea operațiilor de inserare, ștergere și acces. Implementarea operațiilor va fi făcută pentru fiecare structură de date separat.

În acest capitol vom vedea:

- Descrierea structurilor de date uzuale, operațiile permise pe ele și timpii lor de execuție;
- Pentru fiecare structură de date, vom defini o interfață Java conținând protocolul care trebuie implementat;
- Programele complete pentru implementarea acestor structuri.

Scopul este acela de a arăta că specificarea, care descrie funcționalitatea, este independentă de implementare. Nu trebuie să știm *cum* este implementat un anumit lucru, atâta timp cât știm că *este* implementat.

## 10.1 Cum implementăm structurile de date?

Am arătat deja că structurile de date ne permit atingerea unui scop important în programarea orientată pe obiecte: *reutilizarea componentelor*. Așa cum vom vedea mai târziu în acest capitol, structurile de date descrise sunt folosite în multe situații. Odată ce o structură de date a fost implementată, ea poate fi folosită din nou și din nou în aplicații de natură diversă<sup>2</sup>.

Această abordare - separarea interfeței de implementare - este o parte fundamentală a orientării pe obiecte. Cel care folosește structura de date nu trebuie să vadă implementarea ei, ci doar operațiile admisibile. Aceasta ține de partea de ascundere a informației din programarea orientată pe obiecte. O altă parte importantă a programării orientate pe obiecte este *abstractizarea*. Trebuie să proiectăm cu grijă structura de date, deoarece vom scrie programe care folosesc aceste structuri de date fără să aibă acces la implementarea lor. Aceasta va face

<sup>2</sup>De altfel, bibliotecile Java cuprind majoritatea structurilor de date uzuale prezentate în acest capitol: Stack, Queue, Hashtable etc.

Listing 10.1: Interfața `IMemoryCell` pentru clasa `MemoryCell`

```

1 /** Interfața abstractă pentru o celulă de memorie care
2 * stochează un obiect de tip arbitrar */
3 public interface IMemoryCell
4 {
5     /** Intoarce elementul stocat în cadrul celulei */
6     Object read();
7     /** Scrie obiectul x în celulă */
8     void write(Object x);
9 }

```

în schimb ca interfața să fie mai curată, mai flexibilă, și, de obicei, mai ușor de implementat.

Toate structurile de date sunt ușor de implementat dacă nu ne punem problema eficienței. Acest lucru permite să adăugăm componente "ieftine" în program doar pentru depanare. Putem apoi înlocui aceste implementări "ieftine" cu implementări care au o performanță (în timp și/sau în spațiu) mai bună și care sunt adecvate pentru procesarea unei cantități mai mari de informație. Deoarece interfețele sunt fixate, aceste înlocuiri nu necesită practic nici o modificare în programele care folosesc aceste structuri de date.

Vom descrie structurile de date prin intermediul interfețelor. De exemplu, stiva este precizată prin intermediul interfeței `Stack`. Clasa care implementează această interfață va implementa toate metodele specificate în `Stack`, la care se mai pot adăuga anumite funcționalități.

Ca un exemplu, în **Listing 10.1** este descrisă o interfață pentru clasa `MemoryCell`, utilizată în primul volum, la capitolul *Moștenire*, secțiunea *Implementarea de componente generice*. Interfața descrie funcțiile disponibile; clasa concretă trebuie să definească aceste funcții. Implementarea interfeței este prezentată în **Listing 10.2**.

Este important de reținut faptul că structurile de date definite în acest capitol stochează referințe către elementele inserate, și nu copii ale elementelor. Am ales această variantă deoarece este bine ca în structura de date să fie plasate obiecte nemodificabile (cum ar fi `String`, `Integer`, etc.) pentru ca un utilizator extern să nu poată să schimbe starea unui obiect care este înglobat într-o structură de date.

Fiecare dintre structurile prezentate este implementată complet, împreună cu un program de test pentru a verifica corectitudinea implementării structurilor de date. Pentru o mai bună organizare, clasele utilizate în aceste programe sunt împărțite pe pachete, ceea ce înseamnă că fiecare fișier sursă este salvat într-un



Listing 10.2: Implementarea clasei `MemoryCell`

```

1 /** Clasa concreta care implementeaza interfata MemCell */
2 public class MemoryCell implements MemCell
3 {
4     /** Atribut care indica obiectul stocat */
5     private Object storedValue;
6
7     public Object read()
8     {
9         return storedValue;
10    }
11
12    public void write(Object x)
13    {
14        storedValue = x;
15    }
16 }

```

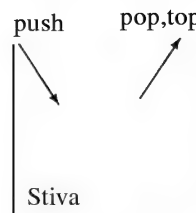
director corespunzător numelui pachetului. Pentru o mai bună înțelegere, considerăm directorul de lucru `c:\javawork` (pentru utilizatorii Windows). Pentru ca programele să funcționeze, acest director trebuie adăugat la variabila sistem `CLASSPATH`. În acest director se vor crea directoarele corespunzătoare pachetelor existente (în cazul nostru, este vorba de pachetele `datastructures` pentru clasele specifice structurilor de date și `exceptions` pentru clasele de excepții), precum și fișierele sursă ale aplicațiilor care folosesc aceste structuri de date (clasele corespunzătoare acestor aplicații de test nu sunt incluse în pachete).

Unele structuri de date folosesc clase în comun cu alte structuri de date. Din acest motiv, implementările claselor respective nu sunt reluate separat pentru fiecare structură de date.

## 10.2 Stive

O stivă este o structură de date în care orice tip de acces este permis doar asupra ultimului element inserat. Comportamentul unei stive este foarte asemănător cu cel al unei grămezi de farfurii. Ultima farfurie adăugată va fi plasată în vârf fiind, în consecință, ușor de accesat, în timp ce farfuriile puse cu mai mult timp în urmă (aflate sub alte farfurii) vor fi mai greu de accesat, putând periclita stabilitatea întregii grămezi. Astfel, stiva este adecvată în situațiile în care avem nevoie să accesăm doar elementul din vârf. Toate celelalte elemente

Figura 10.2: Inserarea în stivă se face prin `push()`, accesul prin `top()`, iar ștergerea prin `pop()`.



Listing 10.3: Clasa de excepții `UnderflowException`

```

1 package exceptions;
2
3 public class UnderflowException extends Exception
4 {
5     public UnderflowException()
6     {
7         super();
8     }
9
10    public UnderflowException(String msg)
11    {
12        super(msg);
13    }
14 }

```

sunt inaccesibile.

Cele trei operații naturale de inserare, ștergere și căutare, sunt denumite în cazul unei stive, `push`, `pop` și `top`. Cele trei operații sunt ilustrate în **Figura 10.2**, iar o interfață Java pentru o stivă abstractă este prezentată în **Listing 10.4**. Interfața declară și o metodă `topAndPop()` care combină două operații, consultare și extragere. În cazul nostru, metodele `pop()`, `top()` și `topAndPop()` pot arunca o excepție `UnderflowException` în cazul în care se încearcă accesarea unui element când stiva este goală. Această excepție va trebui să fie prinsă până la urmă de o metodă apelantă. Clasa `UnderflowException` este definită în **Listing 10.3**, și ea este practic identică cu clasa `Exception`. Important pentru noi este că diferă *tipul*, ceea ce ne permite să prindem *doar* această excepție cu o instrucțiune `try-catch`.

**Listing 10.6** prezintă un exemplu de utilizare a stivei în care se folosește o clasă `StackAr` care reține elementele stivei într-un șir (array). Clasa `StackAr`

## Listing 10.4: Interfață pentru o stivă

```

1 package datastructures;
2
3 import exceptions.*;
4 /** Interfata pentru o stiva. Stiva expune metode pentru
5  * manipularea (adaugarea, stergerea, consultarea)
6  * elementului din varful ei */
7 public interface Stack
8 {
9     public void push(Object x);
10
11     public void pop() throws UnderflowException;
12
13     public Object top() throws UnderflowException;
14
15     public Object topAndPop() throws UnderflowException;
16
17     public boolean isEmpty();
18
19     public void makeEmpty();
20 }

```

este o modalitate de implementare a interfeței `Stack`. `StackAr` folosește un șir pentru a reține elementele din stivă. Mai există și alte modalități de implementare a unei stive. Una dintre acestea se bazează pe liste și implementarea ei este propusă în exercițiul 1. **Listing 10.5** prezintă codul sursă al clasei `StackAr`.

Analizând programul de test din **Listing 10.6** se observă că stiva poate fi folosită pentru a inversa ordinea elementelor. De remarcat un mic artificiu folosit aici: ieșirea din ciclul `for` de la liniile 21-24 se realizează în momentul în care stiva se golește și metoda `topAndPop()` aruncă `UnderflowException` care este prinsă la linia 26. Am recurs la acest artificiu pentru a înțelege mecanismul excepțiilor. Folosirea acestui artificiu în mod curent nu este recomandată, deoarece reduce lizibilitatea codului.

Fiecare operație pe stivă trebuie să ia o cantitate constantă de timp indiferent de dimensiunea stivei, la fel cum accesarea farfuriei din vârful grămezii este rapidă, indiferent de numărul de farfurii din teanc. Accesul la un element oarecare din stivă nu este eficient, de aceea el nici nu este permis de către interfața noastră.

Stiva este deosebit de utilă deoarece sunt multe aplicații pentru care trebuie să accesăm doar ultimul element inserat. Un exemplu ilustrativ este salvarea parametrilor și variabilelor locale în cazul apelului unei alte subrutine.

Implementarea interfeței Stack este realizată folosind șirul de obiecte elements (linia 8 în clasa StackAr). De aici și numele de stivă bazată pe șiruri. Practic, elementele stivei sunt păstrate într-un șir de obiecte, pe care se realizează operațiile expuse de interfață: top(), pop(), push() etc. Stiva are o capacitate inițială de elemente (DEFAULT\_CAPACITY). Dacă pe măsură ce se fac adăugări în stivă, această capacitate este atinsă, atunci stiva își mărește capacitatea cu un număr precizat de elemente (în cadrul metodei increaseStackSize()). Operația este însă ascunsă privirilor utilizatorului, pentru că stiva își mărește dimensiunea fără ca utilizatorul să precizeze sau să aibă cunoștință de acest lucru. Nivelul de "umplere" al stivei este specificat prin intermediul variabilei topPosition, care indică și poziția vârfului stivei.

Listing 10.5: Implementarea unei structuri de tip stivă folosind șiruri

```

1 package datastructures;
2
3 import exceptions.*;
4 /** Implementarea unei stive folosind un sir */
5 public class StackAr implements Stack
6 {
7     /** Sir care retine elementele stivei */
8     private Object[] elements;
9     /**Dimensiunea implicita a stivei*/
10    private static final int DEFAULT_CAPACITY = 10;
11    /** Pozitia varfului stivei */
12    private int topPosition;
13
14    /** Constructor care alocă memorie pentru elements si
15     * initializează varful stivei */
16    public StackAr()
17    {
18        elements = new Object[DEFAULT_CAPACITY];
19        topPosition = -1;
20    }
21
22    /** Adauga elementul x in stiva.*/
23    public void push(Object x)
24    {
25        if (topPosition == elements.length - 1)
26        {
27            increaseStackSize();
28        }
29        elements[++topPosition] = x;
30    }
31
32    /** Mareste dimensiunea (capacitatea) stivei cu

```

```

34   * DEFAULT_CAPACITY atunci cand stiva este plina. */
35  private void increaseStackSize()
36  {
37      Object[] newStack = new Object[elements.length +
38      DEFAULT_CAPACITY];
39
40       //copiem elementele in noua stiva
41      for (int i = 0; i < elements.length; i++)
42      {
43          newStack[i] = elements[i];
44      }
45
46       //elements devine noua stiva
47      elements = newStack;
48  }
49
50   /** Extrage elementul din varful stivei. */
51  public void pop() throws UnderflowException
52  {
53      if (isEmpty())
54      {
55          throw new UnderflowException("Stiva vida.");
56      }
57
58      topPosition--;
59  }
60
61   /** Returneaza elementul din varful stivei
62   * (ultimul adaugat). */
63  public Object top() throws UnderflowException
64  {
65      if (isEmpty())
66      {
67          throw new UnderflowException("Stiva vida.");
68      }
69
70      return elements[topPosition];
71  }
72
73   /** Returneaza elementul din varful stivei si
74   * il elimina apoi din stiva. */
75  public Object topAndPop() throws UnderflowException
76  {
77      if (isEmpty())
78      {
79          throw new UnderflowException("Stiva vida.");
80      }
81
82      return elements[topPosition--];
83  }

```

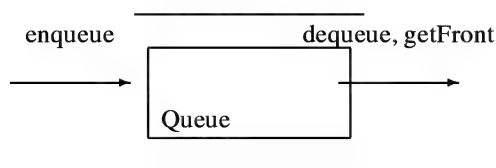
```
84
85  /** Verifica daca stiva e vida. */
86  public boolean isEmpty()
87  {
88      return topPosition == -1;
89  }
90
91  /** Elimina toate elementele din stiva. */
92  public void makeEmpty()
93  {
94      topPosition = -1;
95  }
96 }
```

Listing 10.6: Exemplu de utilizare a stivei. Programul va afișa: Conținutul stivei este 4 3 2 1 0

---

```
1 import datastructures.*;
2 import exceptions.*;
3 /** Clasa de test simpla pentru o stiva, care adauga 5 numere
4 * dupa care le extrage in ordine inversa */
5 public class TestStack
6 {
7     public static void main(String[] args)
8     {
9         Stack s = new StackAr();
10
11         //introducem elemente in stiva
12         for (int i = 0; i < 5; i++)
13         {
14             s.push(new Integer(i));
15         }
16
17         //scoatem elementele din stiva si le afisam
18         System.out.print("Continutul stivei este: ");
19         try
20         {
21             for (; ;)
22             {
23                 System.out.print(s.topAndPop() + " ");
24             }
25         }
26         catch(UnderflowException ue)
27         {
28         }
29     }
30 }
```

Figura 10.3: Modelul unei cozi: adăugarea la coadă se face prin `enqueue()`, accesul prin `getFront()`, ștergerea prin `dequeue()`.



## 10.3 Cozi

O altă structură simplă de date este *coada*. În multe situații este important să avem acces și/sau să ștergem ultimul element inserat. Dar, într-un număr la fel de mare de situații, acest lucru nu numai că nu mai este important, este chiar nedorit. De exemplu, într-o rețea de calculatoare care au acces la o singură imprimantă este normal ca dacă în coada de așteptare se află mai multe documente spre a fi tipărite, prioritatea să îi fie acordată documentului cel mai vechi. Acest lucru nu numai că este corect, dar este și necesar pentru a garanta că documentul nu așteaptă la infinit. Astfel, pe sistemele mari este normal să se folosească cozi de tipărire.

Operațiile fundamentale suportate de cozi sunt:

- *enqueue* - inserarea unui element la capătul cozii;
- *dequeue* - ștergerea primului element din coadă;
- *getFront* - accesul la primul element din coadă.

**Figura 10.3** ilustrează operațiile pe o coadă. Tradițional, metodele `dequeue()` și `getFront()` sunt combinate într-una singură. Prima metodă returnează primul element, după care îl scoate din coadă, în timp ce cea de-a doua returnează primul element fără a-l scoate din coadă.

Implementarea structurii de coadă este asemănătoare până la un punct cu cea a stivei. Coada păstrează elementele într-un șir de obiecte cu o capacitate inițială, iar dacă prin adăugări repetate această capacitate este atinsă, atunci ea va fi mărită automat. Spre deosebire de stivă, coada are doi indici care indică pozițiile de început și de sfârșit ale cozii.

**Listing 10.7** ilustrează interfața pentru o coadă, în timp ce **Listing 10.8** prezintă o implementare a interfeței anterioare, bazată pe șiruri. Spre deosebire de

stivă, în care adăugarea și extragerea unui element au loc la același capăt, în cazul cozii adăugarea are loc la final, dar ștergerea se face de la început. Astfel, pentru a putea utiliza șirul `elements` la întreaga lui capacitate, elementele lui sunt privite “circular”, ca și când ultimul element ar fi legat de primul.

Listing 10.7: Interfață pentru coadă

```

1 package datastructures;
2
3 import exceptions.*;
4 /** Interfata pentru o coada. Expune metode pentru adaugarea
5  * unui element, stergerea primului element, consultarea
6  * primului element */
7 public interface Queue
8 {
9     public void enqueue(Object x);
10
11     public Object getFront() throws UnderflowException;
12
13     public Object dequeue() throws UnderflowException;
14
15     public boolean isEmpty();
16
17     public void makeEmpty();
18 }

```

Listing 10.8: Implementarea unei structuri de tip coadă folosind șiruri

```

1 package datastructures;
2
3 import exceptions.*;
4 /** Implementarea unei cozi folosind un tablou. */
5 public class QueueAr implements Queue
6 {
7     /** Tablou care retine elementele din coada */
8     private Object[] elements;
9     /** Indicele primului element */
10    private int front;
11    /** Indicele ultimului element */
12    private int back;
13    /** Dimensiunea cozii */
14    private int currentSize;
15    /** Numarul de elemente alocat initial pentru coada */
16    private final static int DEFAULT_CAPACITY = 10;
17
18    /** Constructor care alocă memorie pentru elementele cozii
19     * si setează valorile atributelor */
20    public QueueAr()
21    {
22        elements = new Object[DEFAULT_CAPACITY];

```



```

23     makeEmpty();
24 }
25
26 /** Intoarce true daca este vida. */
27 public boolean isEmpty()
28 {
29     return currentSize == 0;
30 }
31
32 /** Adauga elementul x in coada. */
33 public void enqueue(Object x)
34 {
35     if (currentSize == elements.length)
36     {
37         increaseQueueSize();
38     }
39
40     back = increment(back);
41     elements[back] = x;
42     currentSize++;
43 }
44
45 /** Elimina toate elementele din coada. */
46 public void makeEmpty()
47 {
48     currentSize = 0;
49     front = 0;
50     back = -1;
51 }
52
53 /** Extrage primul element din cadrul cozii.
54 *@throws UnderflowException daca coada este goala*/
55 public Object dequeue() throws UnderflowException
56 {
57     if (isEmpty())
58     {
59         throw new UnderflowException("Coadă vida");
60     }
61
62     currentSize--;
63     Object returnValue = elements[front];
64     front = increment(front);
65
66     return returnValue;
67 }
68
69 /** Intoarce primul element din cadrul cozii
70 *@throws UnderflowException daca coada este goala.*/
71 public Object getFront() throws UnderflowException
72 {

```

```

73     if (isEmpty())
74     {
75         throw new UnderflowException("Coadă vida.");
76     }
77
78     return elements[front];
79 }
80
81 /**
82  * Incrementează circular indicele din coadă.*/
83  */
84 private int increment(int x)
85 {
86     if (++x == elements.length)
87     {
88         x = 0;
89     }
90
91     return x;
92 }
93
94 /** Incrementează dimensiunea cozii cu DEFAULT_CAPACITY atunci
95  * când coadă este plină.*/
96 private void increaseQueueSize()
97 {
98     Object[] newQueue = new Object[elements.length +
99     DEFAULT_CAPACITY];
100
101     for (int i = 0; i < elements.length; i++)
102     {
103         newQueue[i] = elements[i];
104         front = increment(front);
105     }
106
107     elements = newQueue;
108     front = 0;
109     back = currentSize - 1;
110 }
111 }

```

Exercițiul 3 propune o a doua modalitate de a implementa această interfață, și anume cu ajutorul listelor, care vor fi prezentate mai târziu în cadrul acestui capitol. **Listing 10.9** prezintă modul de utilizare a cozii. Deoarece operațiile pe o coadă sunt restricționate într-un mod asemănător cu operațiile pe o stivă, este de așteptat ca și aceste operații să fie implementate într-un timp constant. Într-adevăr, toate operațiile pe o coadă pot fi implementate în timp constant,  $O(1)$ .

Listing 10.9: Exemplu de utilizare a cozii. Programul va afișa: Conținutul cozii este: 0 1 2 3 4

```

1 import datastructures.*;
2 import exceptions.*;
3 /** Clasa simpla de test pentru o coada. Se adauga 5 elemente,
4  * dupa care elementele sunt extrase pe rand */
5 public class TestQueue
6 {
7     public static void main(String[] args)
8     {
9         Queue q = new QueueAr();
10
11         //adaugam elemente in coada
12         for (int i = 0; i < 5; i++)
13         {
14             q.enqueue(new Integer(i));
15         }
16
17         //scoatem si afisam elementele din coada
18         System.out.print("Continutul cozii este: ");
19         try
20         {
21             for (; ; )
22             {
23                 System.out.print(q.dequeue() + " ");
24             }
25         }
26         catch (UnderflowException ue)
27         {
28         }
29     }
30 }

```

## 10.4 Liste înlănțuite

Într-o listă înlănțuită elementele sunt reținute discontinuu, spre deosebire de șiruri în care elementele sunt reținute în locații continue. Acest lucru este realizat prin stocarea fiecărui obiect într-un *nod* care conține obiectul și o referință către următorul element în listă, ca în **Figura 10.4**. În acest model se rețin referințe atât către primul (*first*) cât și către ultimul (*last*) element din listă. Un nod al unei liste este implementat în **Listing 10.10**.

Listing 10.10: Un nod al unei liste

```

1 package datastructures;
2 /**
3  * Un nod al listei inlantuite. Clasa este vizibila
4  * doar in pachet (package-friendly), deoarece este
5  * pentru uzul intern al listei.
6  */
7 class ListNode
8 {
9     /** Valoarea continuta de nod.*/
10     Object element;
11     /** Legatura catre nodul urmator.*/
12     ListNode next;
13
14     public ListNode(Object element)
15     {
16         this.element = element;
17         this.next = null;
18     }
19
20     public ListNode(Object element, ListNode next)
21     {
22         this.element = element;
23         this.next = next;
24     }
25 }

```

În orice moment, putem adăuga în listă un nou element  $x$  prin următoarele operații:

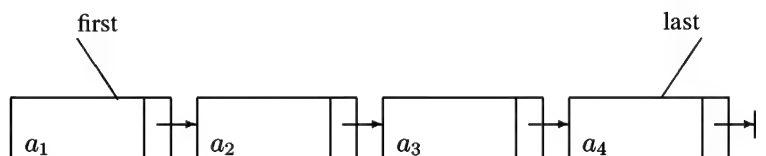
```

last.next = new ListNode(x); //creaza un nod cu continutul x
last = last.next; //nodul creat anterior este ultimul in lista

```

În cazul unei liste înlănțuite un element oarecare nu mai poate fi găsit cu un singur acces. Aceasta este oarecum similar cu diferența între accesarea unei melodii pe CD (un singur acces) și accesarea unei melodii pe casetă (acces secvențial). Deși din acest motiv listele pot să pară mai puțin atractive decât șirurile, există totuși câteva avantaje importante. În primul rând, inserarea unui

Figura 10.4: O listă simplu înlănțuită.



element în mijlocul listei nu implică deplasarea tuturor elementelor de după punctul de inserare. Deplasarea datelor este foarte costisitoare (din punct de vedere al timpului), iar listele înlănțuite permit inserarea cu un număr constant de instrucțiuni de atribuire.

Merită observat că dacă permitem accesul doar la *first*, atunci obținem o stivă, iar dacă permitem inserări doar la *last* și accesări doar la *first*, obținem o coadă. Exercițiile 1 și 3 de la finalul acestui capitol propun exact acest lucru: restricționarea operațiilor pe o listă pentru a obține o stivă respectiv o coadă.

În general, atunci când folosim o listă, avem nevoie de mai multe operații, cum ar fi găsirea sau ștergerea unui element oarecare din listă. Trebuie să permitem și inserarea unui nou element în orice punct. Aceasta este deja mult mai mult decât ne permite o stivă sau o coadă.

Pentru a accesa un element în listă, trebuie să obținem o referință către nodul care îi corespunde. Evident că oferirea unei referințe către un element încalcă principiul ascunderii informației. Trebuie să ne asigurăm că orice acces la listă prin intermediul unei referințe nu periclitează structura listei. Pentru a realiza acest lucru, lista este definită în două părți: o clasă listă și o clasă iterator. Clasa listă va reține elementele propriu-zise ale listei în timp ce iteratorul va oferi o modalitate de a parcurge și modifica elementele listei fără a periclita structura ei. **Listing 10.12** furnizează interfața de bază pentru o listă înlănțuită, oferind și metodele care descriu doar starea listei.

**Listing 10.14** definește o clasă iterator care este folosită pentru toate operațiile de accesare a listei. Pentru a vedea cum funcționează această clasă, să examinăm secvența de cod clasică pentru afișarea tuturor elementelor din cadrul unei structuri liniare. Dacă lista ar fi stocată într-un șir, secvența de cod ar arăta astfel:

```
1 //parcurge sirul a, afisand fiecare element
2 for (int index = 0; index < a.length; ++index)
3 {
4     System.out.println(a[index]);
5 }
```

În Java elementar, codul pentru a itera o listă este:

```
1 //parcurge lista theList de tip List, afisand fiecare element
2 for (ListNode p = theList.first; p != null; p = p.next)
3 {
4     System.out.println(p.data);
5 }
```

Pe de altă parte, trebuie avut în vedere faptul că anumite operații în cadrul listei pot eșua (de exemplu ștergerea unui element inexistent), motiv pentru care este utilizată următoarea clasă `ItemNotFoundException` din **Listing**

**10.11.**

---

**Listing 10.11: Clasa de excepții ItemNotFoundException**

---

```
1 package exceptions ;
2
3 /** Clasa de exceptii care semnaleaza tentativa de a cauta un
4  * element inexistent in cadrul unei structuri de date */
5 public class ItemNotFoundException extends Exception
6 {
7     public ItemNotFoundException ()
8     {
9         super ();
10    }
11
12    public ItemNotFoundException (String msg)
13    {
14        super(msg);
15    }
16 }
```

---

**Listing 10.12: Interfață pentru o listă abstractă**

---

```
1 package datastructures ;
2
3 /** Interfata pentru un tip abstract de lista inlantuita. */
4 public interface List
5 {
6     /** Testeaza daca lista e vida. */
7     boolean isEmpty ();
8
9     /** Sterge toate elementele din lista. */
10    void makeEmpty ();
11 }
```

---

**Listing 10.13: Implementarea unei liste înlanțuite**

---

```
1 package datastructures ;
2
3 /**
4  * Lista simplu inlantuita .
5  * Accesarea elementelor se face cu iteratorul LinkedListItr .
6  */
7 public class LinkedList implements List
8 {
9     ListNode header ;
10
11 }
```

```

12 public LinkedList()
13 {
14     header = new ListNode(null);
15 }
16
17 public boolean isEmpty()
18 {
19     return header.next == null;
20 }
21
22 public void makeEmpty()
23 {
24     header.next = null;
25 }
26 }

```

Listing 10.14: Interfață pentru un iterator abstract de listă

```

1 package datastructures;
2
3 import exceptions.*;
4
5 /**
6  * Interfata iterator pentru parcurgerea elementelor unei
7  * liste inlantuite abstracte (simplu inlantuita, dublu
8  * inlantuita, circulara, etc.)
9  */
10 public interface ListItr
11 {
12     /** Insereaza un element la pozitia curenta.*/
13     void insert(Object x) throws ItemNotFoundException;
14
15     /**
16      * Seteaza pozitia curenta pe elementul x daca
17      * il gaseste in lista.
18      */
19     boolean find(Object x);
20
21     /** Sterge elementul x din lista. */
22     void remove(Object x) throws ItemNotFoundException;
23
24     /** Verifica daca lista a fost parcursa in totalitate. */
25     boolean isInList();
26
27     /** Obtine elementul aflat pe pozitia curenta. */
28     Object retrieve();
29
30     /** Seteaza pozitia inaintea primului element. */
31     void zeroth();
32 }

```

```
33  /** Setează poziția curentă pe primul element. */
34  void first();
35
36  /** Avansează în lista la următorul element. */
37  void advance();
38 }
```

Pornind de la interfața unei liste abstracte, codul pentru implementarea unei liste înlănțuite este dat în **Listing 10.13**, iar implementarea iteratorului unei liste înlănțuite este dată în **Listing 10.15**.

Mecanismul de iterare pe care l-ar folosi limbajul Java ar fi similar cu următoarea secvență:

```
1 //parcurge List, folosind abstractizarea si un iterator
2 ListItr itr = new LinkedListItr(theList);
3
4 for (itr.first(); itr.isInList(); itr.advance())
5 {
6     System.out.println(itr.retrieve());
7 }
```

Inițializarea dinaintea ciclului `for` creează un iterator al listei. Testul de terminare a ciclului folosește metoda `isInList()` definită pentru clasa `LinkedListItr`. Metoda `advance()` trece la următorul nod din cadrul listei. Putem accesa elementul curent prin apelul metodei `retrieve()` definită în `LinkedListItr`. Principiul general este că accesul fiind realizat prin intermediul clasei `ListItr`, securitatea datelor este garantată. Putem avea mai mulți iteratori care să traverseze simultan o singură listă.

Pentru a funcționa corect, clasa `ListItr` trebuie să mențină două obiecte. În primul rând, are nevoie de o referință către nodul curent. În al doilea rând are nevoie de o referință către obiectul de tip `List` pe care îl indică (această referință este inițializată o singură dată în cadrul constructorului).

---

Listing 10.15: Implementarea iteratorului listei înlănțuite

---

```
1 package datastructures;
2
3 import exceptions.*;
4
5 /**
6  * Iterator pentru lista inlantuita.
7  */
8 public class LinkedListItr implements ListItr
9 {
10     /** Referința către lista care va fi iterată */
11     protected LinkedList theList;
12     /** Referința către nodul curent. */
13     protected ListNode current;

```

60



```

14
15 /** Construieste un iterator pe lista list */
16 public LinkedListItr(LinkedList list)
17 {
18     theList = list;
19     current = list.isEmpty() ? list.header : list.header.next;
20 }
21
22 /** Construieste cu parametru de tip List. Daca list
23 * nu refera o instanta LinkedList, se arunca o exceptie
24 * de tipul ClassCastException */
25 public LinkedListItr(List list) throws ClassCastException
26 {
27     this((LinkedList) list);
28 }
29
30 /** Reseteaza iteratorul care va indica pozitia dinaintea
31 * primului element din lista */
32 public void zeroth()
33 {
34     current = theList.header;
35 }
36
37 /** Aadauga obiectul x la pozitia curenta in lista
38 * @throws ItemNotFoundException daca lista este vida */
39 public void insert(Object x) throws ItemNotFoundException
40 {
41     if (current == null)
42     {
43         throw new ItemNotFoundException("Eroare la inserare");
44     }
45
46     ListNode newNode = new ListNode(x, current.next);
47     current.next = newNode;
48     current = current.next;
49 }
50 /** Intoarce true daca obiectul x se afla in lista */
51 public boolean find(Object x)
52 {
53     ListNode itr = theList.header.next;
54
55     while (itr != null && !itr.element.equals(x))
56     {
57         itr = itr.next;
58     }
59
60     if (itr == null)
61     {
62         return false;
63     }

```

```
64
65     current = itr;
66     return true;
67 }
68
69 /** Sterge obiectul x din lista
70 * @throws ItemNotFoundException daca x nu se afla in lista*/
71 public void remove(Object x) throws ItemNotFoundException
72 {
73     ListNode itr = theList.header;
74
75     while (itr.next != null && !itr.next.element.equals(x))
76     {
77         itr = itr.next;
78     }
79
80     if (itr.next == null)
81     {
82         throw new ItemNotFoundException("Stergere esuata");
83     }
84
85     itr.next = itr.next.next; //trecem peste nodul sters
86     current = theList.header;
87 }
88
89 /** Intoarce elementul de pe pozitia curenta */
90 public Object retrieve()
91 {
92     return (isInList() ? current.element : null);
93 }
94
95 /** Intoarce true daca ne aflam in interiorul listei */
96 public boolean isInList()
97 {
98     return (current != null) && (current != theList.header);
99 }
100
101 /** Elementul curent va fi primul element din lista */
102 public void first()
103 {
104     current = theList.header.next;
105 }
106 /** Avanseaza iteratorul pe urmatorul element*/
107 public void advance()
108 {
109     if (current != null)
110     {
111         current = current.next;
112     }
113 }
```

114 }

Iată și exemplul de utilizare a listei:

Listing 10.16: Exemplu de utilizare a listei. Programul va afișa: Conținutul listei: 4 3 2 1 0

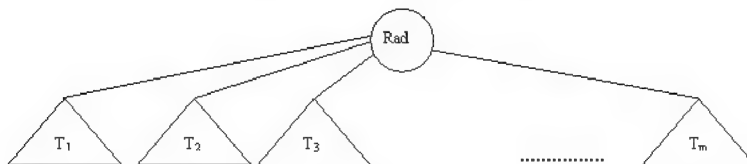
```

1 import datastructures.*;
2 import exceptions.*;
3 /** Clasa simpla pentru testarea unei liste , care adauga 5 numere ,
4  * dupa care afiseaza continutul listei.*/
5 public class TestList
6 {
7     public static void main(String [] args)
8     {
9         List theList = new LinkedList();
10        ListItr itr = new LinkedListItr(theList);
11
12        //se insereaza elemente pe prima pozitie
13        for (int i = 0; i < 5; i++)
14        {
15            try
16            {
17                itr.insert(new Integer(i));
18            }
19            catch (ItemNotFoundException e)
20            {
21            }
22
23            itr.zeroth(); //se trece la inceputul listei
24        }
25
26        System.out.print("Continutul listei: ");
27
28        for (itr.first(); itr.isInList(); itr.advance())
29        {
30            System.out.print(itr.retrieve() + " ");
31        }
32    }
33 }

```

Deși discuția s-a axat pe liste simplu înlănțuite, interfețele din **Listing 10.12** și **Listing 10.14** pot fi folosite pentru oricare tip de listă, indiferent de implementarea pe care o are la bază. Interfața nu precizează faptul că este nevoie de liste simplu înlănțuite.

Figura 10.5: Un arbore generic



## 10.5 Arbori

### 10.5.1 Noțiuni generale

Vom trece acum să studiem cele mai importante structuri neliniare care apar în algoritmii pentru calculatoare: arborii. În general vorbind, structura arborească implică o relație de ramificare între noduri, foarte asemănătoare celei întâlnite la crengile unui arbore din natură.

Una dintre definițiile cele mai răspândite ale arborilor (nu neapărat binari) este următoarea (după D.E. Knuth):

**Definiție:** Un arbore (**Figura 10.5**) este o mulțime finită  $T$  de unul sau mai multe noduri, care are proprietățile:

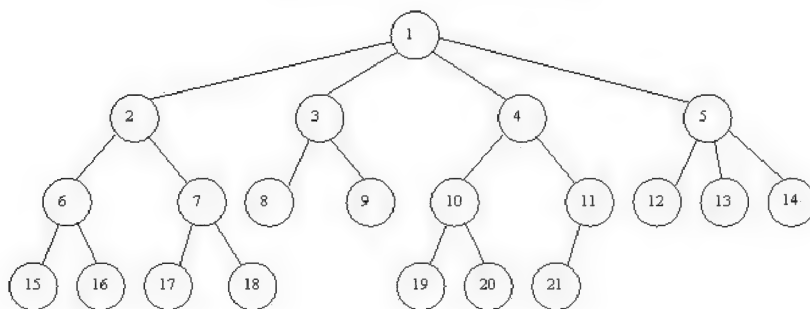
- i) există un nod special, numit rădăcina arborelui;
- ii) toate celelalte noduri din  $T$  sunt repartizate în mulțimi  $T_1, T_2, \dots, T_m$  disjuncte, fiecare mulțime la rândul său fiind un arbore. Arborii  $T_1, T_2, \dots, T_m$  se numesc subarborii rădăcinii.

Se observă că definiția de mai sus este recursivă (recursivitatea este prezentată în capitolul 12): am definit un arbore pe baza unor arbori. Totuși, privind cu atenție definiția ne dăm seama că nu se pune problema circularității, deoarece un arbore cu un singur nod este alcătuit doar din rădăcină, iar arborii cu  $n > 1$  noduri sunt definiți pe baza arborilor cu mai puțin de  $n$  noduri. Există și definiții nerecursive ale arborilor<sup>3</sup>, dar definiția recursivă este mai adecvată, deoarece vom vedea că recursivitatea pare să fie o trăsătură inherentă operațiilor pe structuri arborescente. Caracterul recursiv al arborilor este de altfel prezent și în natură, deoarece mugurii arborilor tineri cresc și se dezvoltă în subarbori, care la rândul lor fac muguri și așa mai departe.

Nodul rădăcină al fiecărui subarbore se numește fiul (sau copilul) rădăcinii, iar rădăcina este tatăl (sau părintele) fiecărui nod rădăcină din subarbori.

<sup>3</sup>De exemplu, în teoria grafurilor, un arbore este definit ca un graf conex și fără cicluri.

Figura 10.6: Un arbore oarecare.



Din definiția recursivă reiese că un arbore este o colecție de  $n$  noduri, dintre care unul este rădăcina, și  $n - 1$  muchii. Faptul că există  $n - 1$  muchii se deduce din observația simplă că fiecare muchie leagă un nod de părintele său și fiecare nod, în afară de rădăcină, are exact un părinte.

În arborele din **Figura 10.6**, rădăcina este 1. Nodul 5 îl are ca părinte pe 1 și are fii 12, 13 și 14. Nodurile care nu au fii se numesc frunze. Frunzele din arborele de mai sus sunt 8, 9, 12, 13, . . . , 21. Nodurile cu același părinte se numesc frați. În mod asemănător se definesc relațiile nepot, bunic etc.

Un drum de la un nod  $n_1$  la un nod  $n_k$  este o secvență de noduri  $n_1, n_2, \dots, n_k$  astfel încât  $n_i$  este tatăl lui  $n_{i+1}$  pentru  $i = 1, k - 1$ . Lungimea unui drum este dată de numărul de muchii ale drumului, adică  $k - 1$ . Pentru oricare nod  $n_i$ , nivelul (adâncimea) nodului este lungimea unicului drum de la rădăcină la  $n_i$ . Astfel, nivelul rădăcinii este 0. În arborele din **Figura 10.6**, nivelul nodului 2 este 1, iar al nodului 15 este 3. Adâncimea unui arbore este definită ca fiind maximul adâncimilor nodurilor. În exemplul nostru, adâncimea arborelui este egală cu adâncimea nodului 15, care este 3.

## 10.5.2 Arbori binari

**Definiție:** Un arbore binar este un arbore în care orice nod are cel mult doi fii.

**Figura 10.7** arată că un arbore binar constă într-o rădăcină și doi subarbori  $T_s$  și  $T_d$ , oricare din ei putând să fie vid (să lipsească).

O proprietate deosebit de importantă a arborilor binari este că adâncimea medie a unui arbore binar cu  $n$  noduri este considerabil mai mică decât  $n$ . Se poate arăta că adâncimea medie a unui arbore binar este proporțională cu  $\sqrt{n}$ , iar adâncimea medie a unui caz particular de arbore binar, arborele binar de

Figura 10.7: Arbore binar generic

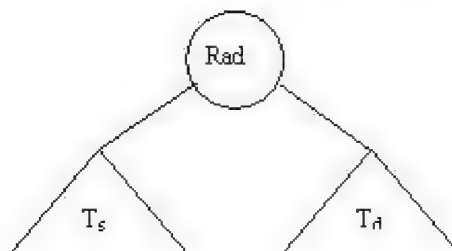
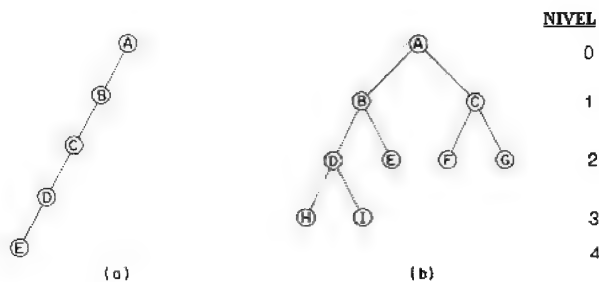


Figura 10.8: Exemple de arbori binari (a) arbore binar degenerat (b) arbore binar nedegenerat



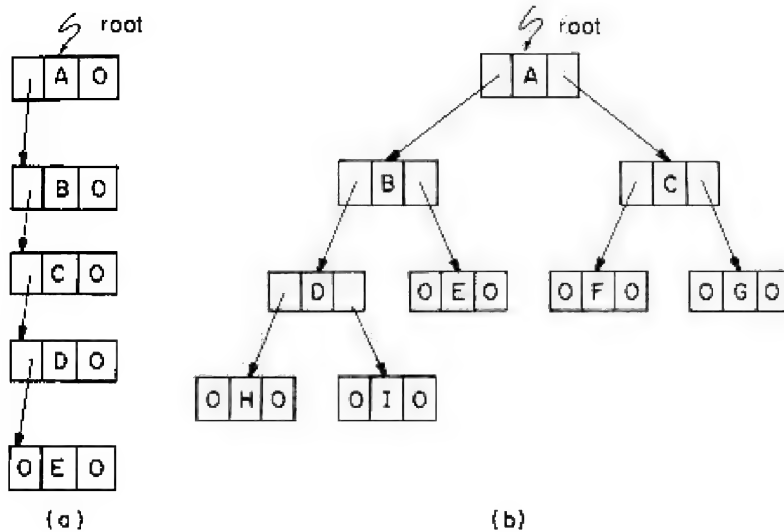
căutare (secțiunea 10.5.3), este proporțională cu  $\log n$ . Din păcate, în cazurile degenerate, adâncimea unui arbore poate fi chiar  $n - 1$ , după cum se vede și în **Figura 10.8(a)**.

Înălțimea (adâncimea) unui arbore binar este foarte importantă, deoarece multe operații definite asupra arborilor (ștergere de nod, inserare de nod etc.) sunt proporționale cu înălțimea arborelui. Din acest motiv, s-a recurs la diferite metode pentru a menține adâncimea unui arbore proporțională cu  $\log n$  în orice situație (arbori AVL, arbori bicolori etc).

### Parcurgerea arborilor binari

După cum reiese și din **Figura 10.7**, putem defini un arbore binar ca fiind o mulțime finită de noduri care este fie vidă, fie este formată dintr-o rădăcină și doi arbori binari. Această definiție sugerează o metodă naturală de reprezentare a arborilor binari: fiecare nod va fi descris de trei componente: `element` - informația utilă a nodului, `left` - o referință către fiul din stânga și `right` -

Figura 10.9: Reprezentarea înlanțuită a arborilor din Figura 10.8



o referință către fiul din dreapta. Dacă un nod nu are fiu în stânga atunci left este null, analog, dacă nu are fiu în dreapta, atunci right este null. Putem astfel defini clasa `BinaryNode`, care reține un nod al unui arbore binar:

```
1 public class BinaryNode
2 {
3     Object element ;
4     BinaryNode left ;
5     BinaryNode right ;
6     ...
7     // constructori si alte metode
8 }
```

Informația utilă din cadrul nodului este reținută de atributul `element`, care este de tip `Object`, deci poate referi o instanță de orice tip. **Figura 10.9** prezintă modul în care se reprezintă înlanțuit arborii binari din **Figura 10.8**. Variabila `root` din **Figura 10.9** este o referință care indică rădăcina arborelui.

Există mai mulți algoritmi pentru manevrarea structurilor arborescente, și o idee care apare de foarte multe ori este noțiunea de parcurgere, de “deplasare” prin arbore. Parcurgerea unui arbore este de fapt o metodă de examinare sistematică a nodurilor arborelui în așa fel încât fiecare nod să fie vizitat o singură

dată. Parcurgerea completă a unui arbore ne oferă o aranjare lineară a nodurilor, și operarea multor algoritmi este simplificată dacă știm care este următorul nod la care ne vom deplasa într-o astfel de secvență, pornind de la un nod dat.

Există trei moduri principale în care un arbore binar poate fi parcurs: nodurile se pot vizita în preordine, inordine și în postordine. Vom descrie aceste trei metode recursiv, ca și definiția arborelui.

Dacă un arbore binar este vid (nu are nici un nod) parcurgerea lui nu presupune nici o operație; în caz contrar parcurgerea comportă trei etape, descrise în tabelul următor:

Preordine	Inordine	Postordine
Se vizitează rădăcina	Se parcurge subarborele stâng	Se parcurge subarborele stâng
Se parcurge subarborele stâng	Se vizitează rădăcina	Se parcurge subarborele drept
Se parcurge subarborele drept	Se parcurge subarborele drept	Se vizitează rădăcina

Pentru arborele din **Figura 10.8** găsim că nodurile în preordine sunt:

A B D H I E C F G

deoarece mai întâi se vizitează rădăcina A, apoi subarborele stâng (B D H I E) și apoi subarborele drept (C F G).

Similar, parcurgerea în inordine are ca rezultat șirul:

H D I B E A F C G

iar parcurgerea în postordine:

H I D E B F G C A

Numele de preordine, inordine și postordine derivă, bineînțeles de la poziția relativă a rădăcinii față de subarbori. O posibilă metodă de a parcurge un arbore în preordine este dată în **Listing 10.17**. Metoda primește ca parametru o referință către rădăcina subarborelui care este vizitat. Evident că la primul apel, parametrul este chiar rădăcina arborelui (referința `root` din **Figura 10.9**). Vom vedea cum se integrează parcurgerea unui arbore în cadrul unei clase complete în paragraful 10.5.3.

Listing 10.17: Metodă generică pentru parcurgerea în preordine a unui arbore binar

```

1 public void preord(BinaryNode t)
2 {
3     if ( t != null )
4     {
5         process( t.element );
6         preord( t.left );
7         preord( t.right );
8     }
9 }

```

68

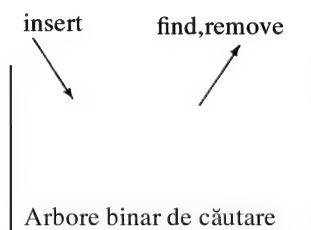


Celelalte două parcurgeri se realizează absolut analog, prin simpla reordonare a instrucțiunilor de parcurgere.

### 10.5.3 Arbori binari de căutare

Una dintre cele mai importante aplicații ale arborilor este utilizarea acestora în probleme de căutare. Proprietatea care face ca un arbore binar să devină un arbore binar de căutare este: pentru oricare nod  $X$  al arborelui toate nodurile din subarboarele stâng sunt mai mici decât  $X$ , și toate nodurile din subarboarele drept sunt mai mari decât  $X$ . Așadar, față de arborii binari obișnuiți, arborii de căutare mai adaugă o relație de ordine între elemente. Pentru a putea fi comparate, nodurile nu vor mai conține obiecte de tip `Object`, ca în paragraful 10.5.2, ci instanțe ale clasei `Comparable`. Arborele binar de căutare este o structură de date în care, pe lângă căutare rapidă, putem adăuga sau șterge eficient elemente. **Figura 10.10** ilustrează operațiile de bază permise asupra unui arbore binar de căutare.

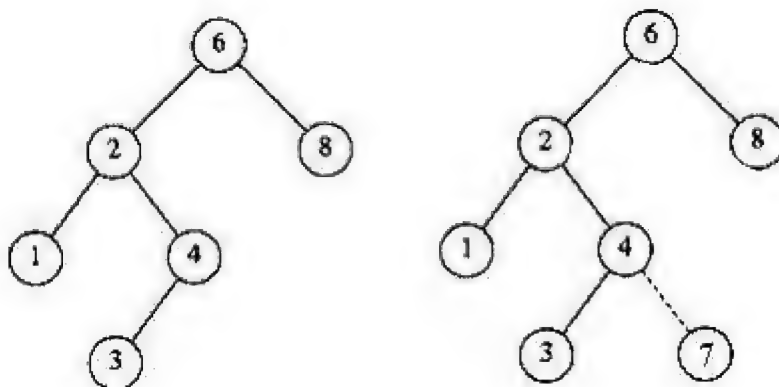
Figura 10.10: Modelul pentru arborele binar de căutare.



De exemplu, arborii din **Figura 10.8** nu sunt arbori binari de căutare, deoarece, în ambele cazuri, în stânga nodului A se află nodul B, care are valoare mai mare decât A (dacă considerăm ordinea alfabetică).

Să studiem cu atenție arborii din **Figura 10.11**. La prima vedere, ambii par să fie arbori binari de căutare; examinând totuși mai minuțios arborele din dreapta, constatăm că nodul 7, deși este mai mare decât rădăcina, 6, se află în stânga ei, ceea ce contravine regulii arborilor de căutare.

Figura 10.11: Doi arbori binari. Doar arborele din stânga este și arbore binar de căutare.



Principalele operații care se realizează asupra arborilor binari de căutare sunt:

- Căutarea unui nod în arbore;
- Găsirea minimului și maximului;
- Inserarea unui nou nod;
- Ștergerea unui nou nod;
- Vidarea arborelui (ștergerea tuturor nodurilor).

O posibilă interfață care descrie aceste operații este prezentată în **Listing 10.19**.

Setul de operații permise este acum extins față de stivă și coadă pentru a permite găsirea unui element arbitrar, alături de inserare și ștergere. Metoda `find()` întoarce o referință către un obiect care este egal (în sensul metodei `compareTo()` din interfața `Comparable`) cu elementul căutat. Dacă nu există nici un element egal cu cel căutat, `find()` aruncă o excepție `ItemNotFoundException`, prezentată în secțiunea anterioară. Aceasta este o decizie de proiectare. O altă posibilitate ar fi fost să întoarcem `null` în cazul în care elementul căutat nu este găsit. Diferența între cele două abordări constă în faptul că metoda noastră îl obligă pe programator să trateze explicit situația în care căutarea nu are succes. În cealaltă situație, dacă am fi întors `null`, și nu s-ar fi făcut verificările necesare, programul ar fi generat un

Listing 10.18: Clasa de excepție DuplicateItemException

```

1 package exceptions;
2
3 /** Clasa de exceptii care semnaleaza adaugarea unui element deja
4  * existent intr-o structura de date */
5 public class DuplicateItemException extends Exception
6 {
7     public DuplicateItemException ()
8     {
9         super();
10    }
11
12    public DuplicateItemException (String msg)
13    {
14        super(msg);
15    }
16 }

```

NullPointerException la prima tentativă de a folosi referința. Din punct de vedere al eficienței, versiunea cu excepții ar putea fi ceva mai lentă, dar este puțin probabil ca rezultatul să fie observabil, cu excepția situației în care codul ar fi foarte des executat în cadrul unor situații în care viteza este critică.

În mod similar, inserarea unui element care deja este în arbore este semnalată printr-o excepție DuplicateItemException (vezi **Listing 10.18**). Există și alte posibile alternative. Una dintre ele constă în a permite noii valori să suprascrie valoarea stocată.

Vom da ca exemplu un arbore de căutare care reține stringuri. Acest lucru este posibil deoarece String implementează interfața Comparable. Implementarea arborelui binar în conformitate cu interfața SearchTree este prezentată în **Listing 10.21**. Pentru a implementa nodurile arborelui binar se utilizează clasa BinaryNode, prezentată în **Listing 10.20**.

Implementarea arborelui binar de căutare este mult mai complexă decât pentru stivă sau coadă și folosește recursivitatea (care va fi prezentată în capitolul 12). Nu vă îngrijorați dacă nu înțelegeți acum toate detaliile de implementare; puteți trece mai departe fără nici o problemă și reveni mai târziu, după ce ați parcurs și înțeles capitolul 12.

Toate elementele din arborele binar ne sunt accesibile prin intermediul rădăcinii sale, care este reținută în atributul root. Să descriem acum pe scurt modul de funcționare al metodelor din clasa BinarySearchTree.

Listing 10.19: Interfața pentru un arbore binar de căutare

```

1 package datastructures;
2
3 import exceptions.*;
4
5 /**
6  * Interfata pentru arbori binari de cautare. Expune metode
7  * pentru adaugarea si stergerea unui element oarecare si
8  * pentru adaugarea si stergerea elementului minim si maxim.
9  */
10 public interface SearchTree
11 {
12     void insert(Comparable x) throws DuplicateItemException;
13     void remove(Comparable x) throws ItemNotFoundException;
14     void removeMin() throws ItemNotFoundException;
15     Comparable find(Comparable x) throws ItemNotFoundException;
16     Comparable findMin() throws ItemNotFoundException;
17     Comparable findMax() throws ItemNotFoundException;
18     boolean isEmpty();
19     void makeEmpty();
20 }

```

**Căutarea unui nod în arbore:** Așa cum le spune și numele, arborii binari de căutare au fost concepuți pentru a se putea căuta informațiile rapid și ușor. Să presupunem că în arborele de căutare din **Figura 10.11** (cel din stânga), dorim să găsim nodul care are cheia 4. Pentru aceasta comparăm valoarea 4 cu rădăcina. Deoarece  $4 < 6$ , nodul căutat se află în subarborele stâng, care are rădăcina 2. Comparăm 4 cu 2. Deoarece  $4 > 2$ , nodul căutat se află în subarborele drept. Comparăm pe 4 cu fiul din dreapta al lui 2, constatăm că valorile sunt egale și ne oprim. Dacă în loc de cheia 4 am fi căutat cheia 5, atunci am fi ajuns în subarborele drept al lui 4, care este vid, deci nodul 5 nu se găsește în arbore.

Căutarea unui nod se realizează cu metoda `find()`, implementată în liniile 75-95 din **Listing 10.21**. Această metodă pornește cu căutarea de la rădăcina arborelui și coboară fie către stânga fie către dreapta (funcție de relația dintre elementul căutat și rădăcina subarborelui) până când elementul căutat este găsit sau s-a "ieșit" din arbore, caz în care se aruncă o `ItemNotFoundException`.

**Găsirea elementului minim și a celui maxim:** Găsirea elementului minim (găsirea maximului se face analog) constă pur și simplu în a coborî în arbore pe partea stângă (liniile 105-108) până se ajunge la frunză. Elementul din frunză este chiar minimul.

Listing 10.20: Clasa care modelează un nod al unui arbore binar

```

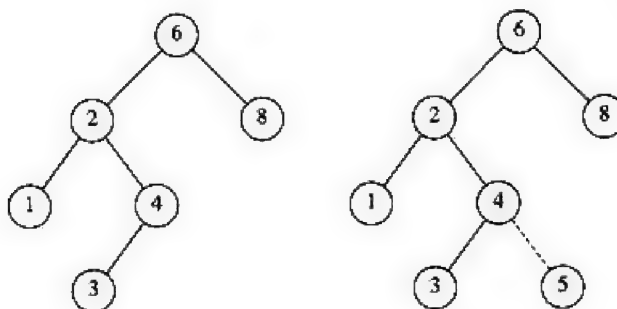
1 package datastructures;
2
3 /**
4  * Reprezinta un nod in arbore. Este pentru uz intern.
5  */
6 class BinaryNode
7 {
8     Comparable element;
9     BinaryNode left;
10    BinaryNode right;
11
12
13    BinaryNode(Comparable e)
14    {
15        element = e;
16        left = null;
17        right = null;
18    }
19
20    BinaryNode(Comparable e, BinaryNode lt, BinaryNode rt)
21    {
22        element = e;
23        left = lt;
24        right = rt;
25    }
26 }

```

**Adăugarea unui nod în arbore:** Adăugarea unui nod în arbore se realizează folosind metoda `insert(Comparable)` din liniile 20-24, care la rândul ei delegă problema către `insert(Comparable, BinaryNode)` de la liniile 129-150. Probabil că vă întrebați de ce a fost nevoie să scindăm operația de adăugare în două metode? Metoda propriu-zisă de ștergere are și un parametru de tip `BinaryNode`, care la început este chiar `root`, rădăcina arborelui. Utilizatorul obișnuit nu are acces la acest atribut, de aceea el poate apela doar metoda `insert(Comparable)`, care fiind o metodă din cadrul clasei, va putea apela la linia 23 `insert(x, root)`. Această tehnică este folosită pentru toate metodele existente în clasa `BinarySearchTree`.

Metoda de adăugare a unui nod în arbore este conceptual destul de simplă. Pentru a insera `x` în arborele cu rădăcina `t`, ne “afundăm” în interiorul arborelui exact ca și în cazul metodei `find()`. Dacă valoarea `x` este găsită în arbore, atunci aruncăm o `DuplicateItemException`. În caz contrar inserăm un nod cu cheia `x`, în ultimul punct din calea care a fost parcursă. **Figura 10.12**

Figura 10.12: Arborele binar de căutare înainte și după inserarea nodului 5.



ne arată ce se petrece. Pentru a insera valoarea 5, traversăm arborele ca și când am realiza procesul de căutare. Când ajungem la nodul cu cheia 4, trebuie să mergem către dreapta, dar nu există subarbore drept, deci 5 nu este în arbore și acesta este locul în care trebuie inserat.

Ca orice metodă recursivă, `insert()` începe la linia 132 cu condiția de terminare:

```
if ( t == null )
```

care este adevărată în momentul în care subarboarele curente este vid. În acest caz se creează pur și simplu un nou nod în care se plasează obiectul `x` și se actualizează rădăcina subarborelui cu nodul curent. În cazul în care subarboarele în care se inserează `x` nu este vid, comparăm `x` cu valoarea din rădăcina arborelui:

```
x.compareTo( t.element )
```

Dacă valoarea din `x` este mai mică, atunci `x` se va insera în subarboarele stâng:

```
t.left = insert(x, t.left) ;
```

altfel, dacă este mai mare se va insera în subarboarele drept:

```
t.right = insert(x, t.right) ;
```

iar dacă valorile sunt egale înseamnă că nodul respectiv exista deja în arbore și se aruncă o `DuplicateItemException`.

Metoda `insert()` conține un aspect de finețe: cum se leagă noul nod inserat (5 în exemplul nostru) de tatăl său (4)? În metoda `insert()` nu pare să existe o secvență de cod care să semnaleze că fiul din dreapta al lui 4 nu mai este `null`, ci este noul nod inserat. Răspunsul se află în modul în care se

realizează apelul recursiv al metodei `insert()`: subarborelui stâng (sau drept) curent i se *atribuie* rezultatul inserării nodului `x` curent. La un moment dat, metoda va crea un nou nod, iar adresa celui nod va fi întoarsă către nodul părinte.

**Ștergerea elementului minim și a celui maxim:** Ștergerea elementului minim constă în găsirea lui, după care se “ridică” subarborele drept în locul lui. Metoda `removeMin()` de la liniile 152-170 realizează exact acest lucru. Observați că aici, pentru variație, ciclul `while` de la liniile 105-108 ale metodei `findMin()` a fost înlocuit cu un apel recursiv. “Ridicarea” subarborelui drept în locul elementului șters se face simplu, prin atribuirea de la linia 166:

```
t = t.right ;
```

Ștergerea elementului maxim se realizează absolut analog.

**Ștergerea unui nod din arbore:** Operația de ștergere a unui nod este cea mai delicată operație pe arborele binar de căutare, deoarece pe lângă eliminarea nodului mai implică și operații suplimentare pentru a păstra structura de arbore binar. În cazul operației de ștergere, după ce am găsit nodul care trebuie șters trebuie să considerăm mai multe posibilități:

- Dacă nodul care trebuie șters este o frunză, el poate fi șters imediat prin înlocuirea legăturii părintelui său cu `null`;
- Dacă nodul care trebuie șters are doar un singur fiu (indiferent că este stâng sau drept), ștergerea lui se face prin ajustarea legăturii părintelui său pentru a-l “ocoli” (**Figura 10.13**);
- Cazul complicat apare atunci când nodul care trebuie șters are doi fii. Strategia generală în acest caz este de a înlocui cheia nodului care trebuie șters cu cea mai mică cheie din subarborele său drept (care este ușor de găsit) după care nodul cu această cheie se șterge folosind metoda `removeMin()` descrisă anterior (**Figura 10.14**).

Ștergerea unui nod din arbore se realizează folosind metoda `remove(Comparable)` din liniile 28-32, care la rândul ei delegă problema către `remove(Comparable, BinaryNode)` de la liniile 172-199. Ca orice metodă recursivă, `remove()` începe la linia 175 cu condiția de terminare:

```
if ( t == null )
```

Figura 10.13: Ștergerea unui nod (4) cu un singur fiu. Legătura 2-3 “ocolește” nodul 4.

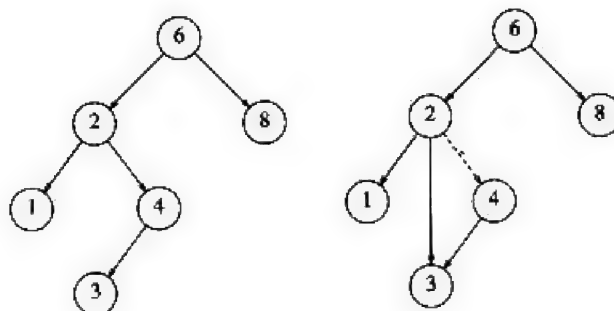
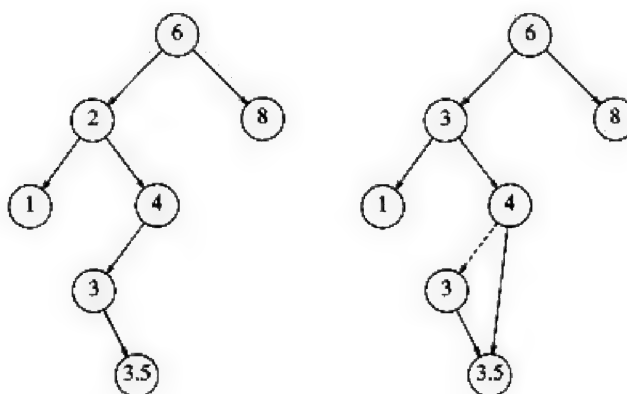


Figura 10.14: Ștergerea unui nod cu doi fii. Cel mai mic nod din subarborele drept al lui 2 este 3. Se înlocuiește 2 cu 3, după care 3 se șterge prin “ocolire”





care este adevărată când subarborele curent este vid. În această situație elementul `x` nu este găsit și se aruncă o `ItemNotFoundException`.

Dacă subarborele nu este vid, se compară elementul care trebuie șters cu rădăcina și, dacă sunt diferite, se coboară fie pe subarborele drept fie pe cel stâng. În cazul în care elementul care trebuie șters a fost găsit (adică `x.compareTo(t.element)` este 0) trebuie să eliminăm nodul respectiv din arbore având însă mare grijă să nu pierdem structura de arbore binar de căutare. Dacă nodul curent are descendenți atât în stânga cât și în dreapta, adică

```
t.left != null && t.right != null
```

este `true`, vom folosi artificul descris anterior: vom aduce în locul nodului șters, cel mai mic nod din subarborele drept (care este evident mai mare decât toate nodurile din subarborele stâng, deci se va păstra structura de arbore de căutare):

```
t.element = findMin(t.right).element ;
```

după care vom apela `removeMin()` pentru a șterge elementul minim din acest subarbore:

```
t.right = removeMin(t.right)
```

Dacă nodul curent nu are descendenți în ambele părți, operația de ștergere este mult mai simplă: pur și simplu se “urcă” subarborele nevid în locul nodului care a fost șters (altfel spus, nodul șters este “ocolit”):

```
t = (t.left != null)? t.left : t.right ;
```

Metodele `isEmpty()` și `makeEmpty()` sunt banale și nu le vom detalia aici.

#### Listing 10.21: Implementarea arborelui binar de căutare

```
1 package datastructures;
2
3 import exceptions.*;
4 /** Arbore binar de cautare alocat inlantuit. Toate
5  * cautarile se fac pe baza metodei compareTo(). Expune
6  * metode pentru inserare, stergere, cautare etc. */
7 public class BinarySearchTree implements SearchTree
8 {
9     /** Referinta catre radacina arborelui */
10    protected BinaryNode root;
11
12    /** Constructor care initializeaza radacina cu null */
13    public BinarySearchTree()
```

```
14 {
15     root = null;
16 }
17
18 /** Aadauga elementul x in arbore
19 * @throws DuplicateItemException daca elementul deja exista.**/
20 public void insert(Comparable x)
21 throws DuplicateItemException
22 {
23     root = insert(x, root);
24 }
25
26 /** Sterge elementul x din arbore.
27 * @throws ItemNotFoundException daca elementul nu se afla
28 * in arbore.**/
29 public void remove(Comparable x)
30 throws ItemNotFoundException
31 {
32     root = remove(x, root);
33 }
34
35 /** Sterge elementul minim din arbore
36 * @throws ItemNotFoundException daca arborele este vid**/
37 public void removeMin() throws ItemNotFoundException
38 {
39     root = removeMin(root);
40 }
41
42 /** Intoarce elementul minim din arbore.
43 * @throws ItemNotFoundException daca arborele este vid**/
44 public Comparable findMin() throws ItemNotFoundException
45 {
46     return findMin(root).element;
47 }
48
49 /** Intoarce elementul maxim din arbore.
50 * @throws ItemNotFoundException daca arborele este vid**/
51 public Comparable findMax() throws ItemNotFoundException
52 {
53     return findMax(root).element;
54 }
55
56 /** Cauta elementul x in arbore.
57 * @throws ItemNotFoundException daca x nu este gasit.**/
58 public Comparable find(Comparable x)
59 throws ItemNotFoundException
60 {
61     return find(x, root).element;
62 }
63
```

```

64  /** Intoarce true daca arborele este vid */
65  public boolean isEmpty()
66  {
67      return root == null;
68  }
69
70  /**Videaza arborele, setand radacina la null */
71  public void makeEmpty()
72  {
73      root = null;
74  }
75
76  protected BinaryNode find(Comparable x, BinaryNode t)
77  throws ItemNotFoundException
78  {
79      while (t != null)
80      {
81          if (x.compareTo(t.element) < 0)
82          {
83              t = t.left;
84          }
85          else if (x.compareTo(t.element) > 0)
86          {
87              t = t.right;
88          }
89          else
90          {
91              return t; //element gasit
92          }
93      }
94
95      throw new ItemNotFoundException("Element negasit.");
96  }
97
98  protected BinaryNode findMin(BinaryNode t)
99  throws ItemNotFoundException
100 {
101     if (t == null)
102     {
103         throw new ItemNotFoundException("Element negasit");
104     }
105
106     while (t.left != null)
107     {
108         t = t.left;
109     }
110
111     return t;
112 }
113

```

```

114 protected BinaryNode findMax(BinaryNode t)
115 throws ItemNotFoundException
116 {
117     if (t == null)
118     {
119         throw new ItemNotFoundException("Element negasit");
120     }
121
122     while (t.right != null)
123     {
124         t = t.right;
125     }
126
127     return t;
128 }
129
130 protected BinaryNode insert(Comparable x, BinaryNode t)
131 throws DuplicateItemException
132 {
133     if (t == null)
134     {
135         t = new BinaryNode(x, null, null);
136     }
137     else if (x.compareTo(t.element) < 0)
138     {
139         t.left = insert(x, t.left);
140     }
141     else if (x.compareTo(t.element) > 0)
142     {
143         t.right = insert(x, t.right);
144     }
145     else
146     {
147         throw new DuplicateItemException("Elementul exista deja");
148     }
149
150     return t;
151 }
152
153 protected BinaryNode removeMin(BinaryNode t)
154 throws ItemNotFoundException
155 {
156     if (t == null)
157     {
158         throw new ItemNotFoundException("Element negasit");
159     }
160
161     if (t.left != null)
162     {
163         t.left = removeMin(t.left);

```

```

164     }
165     else
166     {
167         t = t.right;
168     }
169
170     return t;
171 }
172
173 protected BinaryNode remove(Comparable x, BinaryNode t)
174 throws ItemNotFoundException
175 {
176     if (t == null)
177     {
178         throw new ItemNotFoundException("Element negasit");
179     }
180
181     if (x.compareTo(t.element) < 0)
182     {
183         t.left = remove(x, t.left);
184     }
185     else if (x.compareTo(t.element) > 0)
186     {
187         t.right = remove(x, t.right);
188     }
189     else if (t.left != null && t.right != null)
190     {
191         t.element = findMin(t.right).element;
192         t.right = removeMin(t.right);
193     }
194     else
195     {
196         t = (t.left != null) ? t.left : t.right;
197     }
198
199     return t;
200 }
201
202 }

```

**Listing 10.22** prezintă modul în care arborele binar de căutare poate fi folosit pentru obiecte de tip `String`.

Interfața `SearchTree` mai are două metode suplimentare: una pentru a găsi cel mai mic element și una pentru a găsi cel mai mare element. Se poate arăta că transpirând un pic mai mult se poate găsi foarte eficient și cel mai mic al  $k$ -lea element, pentru oricare  $k$  trimis ca parametru.

Să vedem care sunt timpii de execuție pentru operațiile pe un arbore binar de căutare. Este normal să sperăm că timpii de execuție pentru `find()`,

`insert()` și `remove()` să fie logaritmici, deoarece aceasta este valoarea pe care o vom obține pentru căutarea binară (paragraful 12.3). Din nefericire, pentru cea mai simplă implementare a arborelui binar de căutare, acest lucru nu este adevărat. Timpul mediu de execuție este într-adevăr logaritmice, dar în cazul cel mai nefavorabil (când arborele este “dezechilibrat”) timpul de execuție este  $O(n)$ , caz care apare destul de frecvent. Totuși, prin aplicarea anumitor trucuri de algoritmică se pot obține anumite structuri mai complexe (arbori bicolori) care au într-adevăr un cost logaritmice pentru fiecare operație.

Listing 10.22: Model de program care utilizează arbori de căutare. Programul va afișa: Găsit Cristi;Alina nu a fost găsită

```

1 import datastructures.*;
2 import exceptions.*;
3 /** Clasa simpla de test pentru arborii binari de cautare.*/
4 public class TestSearchTree
5 {
6     public static void main(String[] args)
7     {
8         SearchTree t = new BinarySearchTree();
9
10        String result = null;
11
12        try
13        {
14            t.insert("Cristi");
15        }
16        catch(DuplicateItemException e)
17        {
18        }
19
20        try
21        {
22            result = (String) t.find("Cristi");
23
24            System.out.print("Gasit " + result + " ");
25        }
26        catch(ItemNotFoundException e)
27        {
28            System.out.print("Cristi nu a fost gasit ");
29        }
30
31        try
32        {
33            result = (String) t.find("Alina");
34
35            System.out.print("Gasit " + result + " ");
36        }
37        catch(ItemNotFoundException e)

```

```

38     {
39         System.out.print("Alina nu a fost gasita;");
40     }
41 }
42 }

```

Ce putem spune despre operațiile `findMin()` și `findMax()`? În mod cert, aceste operații necesită un timp constant în cazul căutării binare, deoarece implică doar accesarea unui element indicat. În cazul unui arbore binar de căutare aceste operații iau același timp ca o căutare obișnuită, adică  $O(\log n)$  în cazul mediu și  $O(n)$  în cazul cel mai nefavorabil.

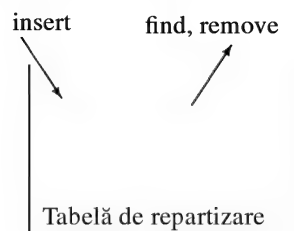
## 10.6 Tabele de repartizare

Există foarte multe aplicații care necesită o căutare dinamică bazată doar pe un nume. O aplicație clasică este *tabela de simboluri* a unui compilator. Pe măsură ce compilează programul, compilatorul trebuie să rețină numele (împreună cu tipul, durata de viață, locația de memorie) tuturor identificatorilor care au fost declarați. Atunci când vede un identificator în afara unei instrucțiuni de declarare, compilatorul verifică să vadă dacă acesta a fost declarat. Dacă a fost, compilatorul verifică informația adecvată din tabela de simboluri.

Având în vedere faptul că arborele binar de căutare permite acces logarithmic la obiecte cu denumiri oarecare, de ce am avea nevoie de o altă structură de date? Răspunsul este că arborele binar de căutare poate să dea un timp de execuție liniar pentru accesul unui element, iar pentru a ne asigura de cost logarithmic este nevoie de algoritmi mult mai sofisticati.

*Tabela de repartizare* (engl. *hashtable*) este o structură de date care evită timpul de execuție liniar, ba mai mult, suportă aceste operații în timp (mediu) constant. Astfel, timpul de acces la un element din tabelă nu depinde de numărul de elemente care sunt în tabelă. În același timp, tabela de repartizare nu folosește neapărat alocarea înlănuțită (ca arborele binar). Aceasta face ca tabela de repartizare să fie rapidă în practică. Un alt avantaj față de arborele binar de căutare este că elementele stocate în tabela de repartizare nu trebuie să implementeze interfața `Comparable`. Acum probabil că vă întrebați: bine, dar dacă tabela de repartizare este atât de eficientă, de ce se mai folosesc arbori binari? Răspunsul este că arborii binari, deși au operații de inserare și acces care se execută în timp logarithmic dispun de operații pe care tabele de repartizare nu le suportă. Astfel, nu este posibil să parcurgem eficient elementele dintr-o tabelă de repartizare. Practic, tabelele de repartizare oferă un suport eficient pentru numai trei operații: adăgarea unui element, ștergerea unui element și căutarea unui element. Parcurgerea elementelor, calculul minimului sau maximului nu sunt suportate.

Figura 10.15: Modelul pentru tabela de repartizare: orice element etichetat poate fi adăugat sau șters în timp practic constant.



Listing 10.23: Interfața pentru tabele de repartizare

```

1 package datastructures;
2
3 import exceptions.*;
4 /** Interfata expusa de o tabela de repartizare. Oferă
5  * operatii de adaugare, stergere, cautare si golire. */
6 public interface HashTable
7 {
8     void insert(Hashable x);
9     void remove(Hashable x) throws ItemNotFoundException;
10    Hashable find(Hashable x) throws ItemNotFoundException;
11    void makeEmpty();
12 }

```

Operațiile permise asupra unei tabele de repartizare sunt date în **Figura 10.15**, iar o interfață este prezentată în **Listing 10.23**. În acest caz, inserarea unui element care este duplicat nu generează o excepție, deoarece elementul va fi înlocuit cu noua valoare. Aceasta este o alternativă la metoda pe care am aplicat-o în cazul arborilor binari de căutare. Tabela de repartizare funcționează doar pentru elemente care implementează interfața `Hashable`<sup>4</sup>. Interfața `Hashable` (**Listing 10.24**) expune o *funcție de repartizare*, care asociază obiectului de tip `Hashable` un număr întreg.

Fiecare dintre elementele conținute de o tabelă de repartizare va fi asociat cu un *număr*, pentru a permite accesul în timp constant la elementul respectiv. Acest număr este calculat de metoda `hash()` expusă de interfața `Hashable`. Vom vedea mai târziu cum se face accesul la element pe baza acestui număr

<sup>4</sup>În limba engleză, tabelele de repartizare se numesc "Hashtable", de aceea un element care poate fi repartizat se numește "Hashable".



care este calculat într-un timp constant.

Elementele dintr-o tabelă de repartizare trebuie să redefinească și metoda `equals()`. **Listing 10.25** arată cum clasa `MyString` implementează interfața `Hashable` prin implementarea metodelor `hash()` și `equals()`.

Listing 10.24: Interfața `Hashable`

```
1 package datastructures;
2 /** Descrie un element care poate fi stocat într-o
3  * tabela de repartizare */
4 public interface Hashable
5 {
6     /** Asociaza un numar intreg instantei Hashable */
7     int hash(int tableSize);
8 }
```

Listing 10.25: Clasa `MyString`

```
1 import datastructures.*;
2 import exceptions.*;
3
4 /** Exemplu de clasa care implementeaza interfata Hashable, deci
5  * instantele ei pot fi stocate într-o tabela de repartizare. */
6 public class MyString implements Hashable
7 {
8     private String value;
9
10
11     public MyString(String value)
12     {
13         this.value = value;
14     }
15
16     public String toString()
17     {
18         return value;
19     }
20
21     public boolean equals(Object c)
22     {
23         return value.equals(((MyString) c).value);
24     }
25
26     public int hash(int tableSize)
27     {
28         return QuadraticProbingTable.hash(value, tableSize);
29     }
30 }
```

Pentru a calcula numărul asociat unui obiect de tip `MyString` în cadrul

tabelei de repartizare, metoda `hash()` apelează la rândul ei o metodă statică a clasei `QuadraticProbingTable`, care va fi definită mai târziu (vezi **Listing 10.28**).

Listing 10.26: Un element al tabelului de repartizare

```

1 package datastructures;
2
3 /** Clasa care descrie un element dintr-o tabela de repartizare.
4  * Elementul este descris de o instanta Hashable si o valoare
5  * booleana care indica daca elementul este activ. */
6 class HashEntry
7 {
8     Hashable element;
9     boolean isActive;
10
11     public HashEntry (Hashable e)
12     {
13         element = e;
14         isActive = true;
15     }
16
17     public HashEntry (Hashable e, boolean i)
18     {
19         element = e;
20         isActive = i;
21     }
22 }

```

Interfața `HashTable` expune operațiile pe care le suportă structura de date: inserare, ștergere, căutare și eliminarea tuturor elementelor. Unele dintre aceste operații pot arunca excepția `ItemNotFoundException`, care a fost prezentată în secțiunea 10.4.

Să vedem acum cum se implementează tabela de repartizare. Vom defini mai întâi un element al tabelului de repartizare, reprezentat de clasa `HashEntry` din **Listing 10.26**. `HashEntry` modelează un element din tabela de repartizare. Practic, tabela de repartizare va conține elemente de tipul `HashEntry`. Pentru fiecare element din tabelă este necesar să specificăm valoarea lui (atributul `element`, de tip `Hashable`) și dacă elementul este activ (atributul `isActive`). Trebuie menționat că ștergerea unui element din tabela de repartizare nu implică eliminarea lui fizică din structură, ci doar "dezactivarea" lui, prin setarea atributului `isActive` la `false` (paragraful 10.6.1).

Implementarea propriu-zisă a tabelului de repartizare constă din următoarele două clase: `ProbingHashTable` și `QuadraticProbingTable`.

Listing 10.27: Clasa `ProbingHashTable`

```

1 package datastructures;
2
3 import exceptions.*;
4 /** Implementeaza o tabela cu repartizare inchisa, in care
5  * elementele tabelii de repartizare se retin intr-un tablou */
6 public abstract class ProbingHashTable
7 implements HashTable
8 {
9     protected HashEntry[] elements;
10    private int currentSize;
11    private static final int DEFAULT_TABLE_SIZE = 11;
12
13    /** Calculeaza pozitia unui element in cadrul tabelii */
14    protected abstract int findPos(Hashable x);
15
16    /** Aloca memorie si initializeaza elementele tabloului */
17    public ProbingHashTable()
18    {
19        elements = new HashEntry[DEFAULT_TABLE_SIZE];
20        makeEmpty();
21    }
22
23    /** Goleste tabela de repartizare, si elibereaza toate
24     * referintele din tabloul elements */
25    public void makeEmpty()
26    {
27        currentSize = 0;
28
29        for (int i = 0; i < elements.length; i++)
30        {
31            elements[i] = null;
32        }
33    }
34
35    /** Intoarce elementul cu cheia x.hash()
36     * @throws ItemNotFoundException daca elementul nu e gasit */
37    public Hashable find(Hashable x)
38    throws ItemNotFoundException
39    {
40        int currentPos = findPos(x);
41
42        assertFound(currentPos, "Element negasit");
43
44        return elements[currentPos].element;
45    }
46
47    /** Verifica daca elementul de pe pozitia currentPos
48     * exista si este activ.
49     * @throws ItemNotFoundException in caz contrar */

```

```

50 private void assertFound(int currentPos , String message)
51 throws ItemNotFoundException
52 {
53     if (elements[currentPos] == null ||
54         elements[currentPos].isActive == false)
55     {
56         throw new ItemNotFoundException(message);
57     }
58 }
59
60 /** Sterge elementul x din tabela de repartizare.
61     * @throws ItemNotFoundException daca x nu e gasit. */
62 public void remove(Hashable x)
63 throws ItemNotFoundException
64 {
65     int currentPos = findPos(x);
66
67     assertFound(currentPos , "Element negasit");
68
69     elements[currentPos].isActive = false;
70 }
71
72 /** Adauga elementul x tabelii de repartizare. Daca
73     * se atinge gradul de incarcare maxima adims se
74     * dubleaza numarul de elemente din elements */
75 public void insert(Hashable x)
76 {
77     int currentPos = findPos(x);
78
79     elements[currentPos] = new HashEntry(x, true);
80
81     if (++currentSize < elements.length / 2)
82     {
83         return;
84     }
85
86     //dimensiunea tabelii este prea mica
87     //si pot aparea conflicte, deci vom dubla numarul de elemente
88     HashEntry[] oldElements = elements;
89
90     //creaza un tabel de dimensiune dubla
91     elements = new HashEntry[oldElements.length];
92     currentSize = 0;
93
94     //copie vechiul tabel in cel nou
95     for(int i = 0; i < oldElements.length; i++)
96     {
97         if (oldElements[i] != null && oldElements[i].isActive)
98         {
99             insert(oldElements[i].element);

```

```

100     }
101   }
102 }
103
104 /** Asociază un număr unic unei chei de tip String
105  * cu valoarea între 0 și tableSize-1. */
106 public static int hash(String key, int tableSize)
107 {
108     int hashVal = 0;
109
110     for (int i = 0; i < key.length(); i++)
111     {
112         hashVal = 37 * hashVal + key.charAt(i);
113     }
114
115     hashVal %= tableSize;
116
117     if (hashVal < 0)
118     {
119         hashVal += tableSize;
120     }
121
122     return hashVal;
123 }
124 }

```

ProbingHashTable este clasa cea mai importantă în implementarea unei tabele de repartizare. Ea implementează interfața HashTable și definește toate operațiile expuse de aceasta. Elementele din tabela de repartizare sunt reținute într-un șir de obiecte de tip HashEntry (atributul `elements`). Structura mai specifică de asemenea și numărul de elemente care sunt stocate la momentul curent (atributul `currentSize`).

Inserarea unui element în tabela de repartizare presupune asocierea unui număr unic respectivului element, prin intermediul metodei `hash()`. Acest număr devine indicele elementului în șirul `elements`, motiv pentru care accesul în timp constant este asigurat. Practic, prin inserarea elementelor în tabela de repartizare, șirul `elements`, care reține aceste elemente, va fi populat într-un mod discontinuu (elementele nu se vor afla pe poziții succesive, pentru că poziția lor depinde de valoarea lor). Când se va face căutarea elementului, se va realiza operația de `hash()` prin care se obține numărul asociat, și se va accesa direct elementul de pe poziția calculată. Evident, accesul este în timp constant.

Metoda `hash()` calculează numărul asociat unui element pe baza unui algoritmul simplu. Se realizează o sumă de produse pe baza codului *Unicode* al fiecărui caracter din cheia elementului care va fi inserat în tabelă, iar în final se consideră restul împărțirii acestei valori la dimensiunea tabelei, pentru că

Listing 10.28: Clasa QuadraticProbingTable

```

1 package datastructures;
2 /** Tabela cu repartizare patratica inchisa. Implementeaza metoda
3  * abstracta findPos() mostenita de la ProbingHashTable */
4 public class QuadraticProbingTable extends ProbingHashTable
5 {
6     protected int findPos(Hashable x)
7     {
8         int collisionNum = 0;
9         int currentPos = x.hash(array.length);
10
11         while (array[currentPos] != null &&
12             !array[currentPos].element.equals(x))
13         {
14             collisionNum++;
15             currentPos += 2 * collisionNum - 1;
16
17             if (currentPos >= array.length)
18             {
19                 currentPos -= array.length;
20             }
21         }
22
23         return currentPos;
24     }
25 }
26 }

```

numărul asociat elementului trebuie să fie mai mic decât dimensiunea tabelului (numărul nu este altceva decât un indice al șirului `elements`).

### 10.6.1 Tratarea coliziunilor

Este important să observăm faptul că metoda `hash()` nu ne asigură de faptul că obiecte distincte vor avea repartizări distincte. De fapt, există șanse considerabile ca pentru două elemente diferite, metoda `hash()` să returneze același număr. Aceasta este o situație inacceptabilă, pentru că nu putem avea două elemente care să fie pe aceeași poziție în șirul `elements`. Din acest motiv, avem nevoie de o strategie care să elimine coliziunile care pot să apară. Strategia va determina dacă numărul asociat de metoda `hash()` este deja ocupat și va oferi un alt număr care nu a fost alocat anterior. Această strategie va fi utilizată atât în operația de adăugare, cât și în cea de căutare a elementului.

Există mai multe strategii disponibile, care se bazează pe căutarea unui spațiu neocupat în șirul `elements`. De aici provine și numele clasei `ProbingHashTable`. Cu alte cuvinte, în caz de coliziune, se *probează* elementele din șirul `elements`, până se descoperă unul neocupat.

Pentru a oferi suport pentru implementarea oricărei strategii, am decis ca `ProbingHashTable` să declare abstractă metoda care tratează situațiile de coliziune (este vorba despre metoda `findPos()`), motiv pentru care și clasa devine abstractă, urmând ca fiecare strategie să definească această metodă într-o clasă derivată separată.

Cea mai simplă strategie pentru evitarea coliziunii este *probarea* liniară, cu alte cuvinte căutarea secvențială (element cu element) în șir, până se găsește o celulă goală, care să fie alocată elementului în cauză. Performanța acestei strategii nu este ridicată, mai ales în situația în care avem o tabelă de dimensiuni mari.

O strategie mai performantă este cea *pătratică*, în care căutarea nu se face liniar ( $i, i + 1, i + 2, i + 3$  etc.), ca în exemplul anterior, ci în "salturi" mai mari de tipul  $i, i + 1^2, i + 2^2, i + 3^2$  etc. Acesta este motivul pentru care clasa care definește metoda `findPos()` poartă numele de `QuadraticProbingTable` (*quadratic* == pătratic). Deoarece în cazul unui conflict, elementul nu va fi pus pe poziția dată de funcția de repartizare ci pe locația liberă găsită prin probare, căutarea unui element va trebui făcută tot prin probare, deci tot utilizând metoda `findPos()`, apelată la linia 40 în cadrul metodei `find()` din clasa `ProbingHashTable`. Astfel, căutarea se va opri fie când elementul este găsit, fie când s-a ajuns la o locație goală în `elements`. O altă observație importantă este că ștergerea unui element nu va putea fi realizată prin setarea poziției care îi corespunde în `elements` la `null`, deoarece aceasta va împiedica găsirea elementelor cu aceeași valoare a funcției de repartizare care au fost așezate prin probare după elementul șters. Acesta este motivul pentru care ștergerea unui element se face prin setarea atributului `isActive` la `false`.

**Listing 10.29** prezintă un exemplu de utilizare al tabeli de repartizare.

Există foarte multe modalități de a implementa tabelele de repartizare. Noi am prezentat aici doar una singură, numită repartizare pătratică închisă. Pentru o detalii referitoare la acest subiect recomandăm excelenta lucrare *Data Structures* (vezi [Tomescu]).

Listing 10.29: Exemplu de program care folosește tabele de repartizare; programul va afișa: Gasit Marcel

```
1 import datastructures.*;
2 import exceptions.*;
```

```

3 /** Clasa simpla de test pentru tabelele de repartizare. */
4 public class TestHashTable
5 {
6     public static void main(String [] args)
7     {
8         HashTable h = new QuadraticProbingTable();
9
10        MyString result = null;
11
12        h.insert(new MyString("Marcel"));
13
14        try
15        {
16            result = (MyString) h.find(new MyString("Marcel"));
17            System.out.println("Gasit " + result);
18        }
19        catch(ItemNotFoundException e)
20        {
21            System.out.println("Marcel nu a fost gasit");
22        }
23    }
24 }

```

## 10.7 Cozi de prioritate

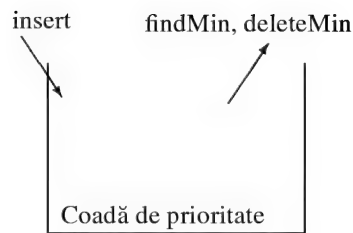
Să pornim de la o problemă concretă extrem de simplă: ordinea de tipărire a documentelor la o imprimantă partajată, în cadrul unei rețele, de un număr mare de utilizatori. În mod obișnuit documentele trimise unei imprimante sunt așezate, într-o coadă simplă, dar aceasta nu este întotdeauna cea mai bună variantă. De exemplu, un document poate să fie deosebit de important, deci el ar trebui executat imediat ce imprimanta este disponibilă. De asemenea, dacă imprimanta a terminat de tipărit un document, iar în coadă se află câteva documente având 1-2 pagini și un document având 100 de pagini, ar fi normal ca documentul lung să fie tipărit ultimul, chiar dacă nu este ultimul document trimis.

Analog, în cazul unui sistem multiutilizator, sistemul de operare trebuie să decidă la un moment dat care proces, dintre cele existente, trebuie să fie executat. În general, un proces poate să se execute doar o perioadă de timp fixată. Și aici este normal ca procesele care au nevoie de un timp foarte mic să aibă prioritate.

Dacă vom atribui fiecărei sarcini câte un număr, atunci numărul mai mic (pagini tipărite, resurse folosite) va indica o prioritate mai mare. Astfel, vom dori să accesăm cel mai mic element dintr-o colecție de elemente și să îl ștergem



Figura 10.16: Modelul pentru coada de prioritate. Doar elementul minim este accesibil.



Listing 10.30: Interfața PriorityQueue

```

1 package datastructures;
2
3 import exceptions.*;
4 /**
5  * Interfața care descrie operațiile expuse de o coadă
6  * de prioritate.
7  */
8 public interface PriorityQueue
9 {
10  /** Adauga elementul x in coada de prioritate */
11  void insert(Comparable x) ;
12  /** Sterge si intoarce elementul minim. */
13  Comparable deleteMin() throws UnderflowException ;
14  /** Intoarce elementul minim din coada */
15  Comparable findMin() throws UnderflowException ;
16  /** Goleste coada de prioritate */
17  void makeEmpty() ;
18  /** Intoarce true daca coada este vida */
19  boolean isEmpty() ;
20
21 }
  
```

din cadrul colecției. Acestea sunt operațiile `findMin` și `deleteMin`. Structura de date care oferă o implementare foarte eficientă a acestor operații se numește *coadă de prioritate*. **Figura 10.16** ilustrează operațiile fundamentale pentru coada de prioritate.

Interfața unei cozi de prioritate este descrisă în **Listing 10.30**.

Clasa de excepții `UnderflowException` a fost prezentată în secțiunea 10.2 și are rolul de a semnaliza situația în care s-a încercat accesarea unui element

Figura 10.17: Un arbore binar complet. Se observă că toate nodurile au exact doi fii, mai puțin nodurile de pe ultimul nivel (D, E, F și G) pentru care frunzele (H, I și J) sunt completate de la stânga la dreapta.

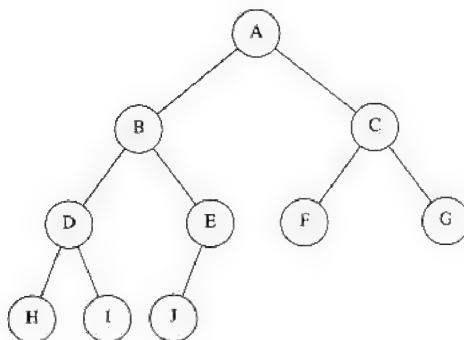


Figura 10.18: Reprezentarea arborelui din figura 10.17 utilizând un tablou. Nodurile arborelui sunt introduse în tablou în ordinea naturală a citirii lor de sus în jos și de la stânga la dreapta. Prima locație rămâne necompletată.

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

într-o coadă de priorități goală.

### 10.7.1 Ansamblu

Implementarea cozii de priorități o prezentăm sub forma unui ansamblu (engl. *heap*, tradus în unele lucrări prin termeni amuzanți cum ar fi *movilă* sau *grămadă*). Ansamblele sunt arbori binari compleți. Cu alte cuvinte, fiecare nod are câte doi fii, mai puțin pe ultimul nivel, unde se poate întâmpla ca un nod să nu aibă fii, dar în această situație, nici un nod aflat la dreapta sa nu va avea fii.

O observație importantă pe care o putem face este că dat fiind faptul că un arbore binar complet este atât de regulat, elementele sale pot fi cu ușurință reprezentate într-un șir, fără a fi necesar să utilizăm alocarea înlănțuită din paragraful 10.5.3! Tabloul din **Figura 10.17** corespunde arborelui din **Figura 10.18**.

Problema care se pune acum este: cum găsim în reprezentarea sub formă de șir a arborelui care sunt fii și care este părintele unui anumit nod? Privind cu atenție cele două figuri se observă cu ușurință că întotdeauna fii unui nod care este pe poziția  $i$  în tablou se vor afla pe pozițiile  $2*i$  și  $2*i+1$ . Pe poziția  $2*i$  se află fiul stâng, iar pe poziția  $2*i+1$  se află fiul drept. În consecință, părintele unui nod aflat de poziția  $i$  se va afla pe poziția  $i/2$  (împărțire întreagă). Astfel, nu numai că nu este nevoie de alocare înlănțuită pentru a reprezenta un ansamblu, ci, mai mult, operațiile de traversare vor fi extrem de rapide (vezi capitolul 2 din primul volum, subcapitolul *Operatori la nivel de bit*)! Singura problemă cu această implementare este că avem nevoie de o aproximare a numărului maxim de elemente din ansamblu pentru a stabili dimensiunea șirului în care vor fi memorate elementele.

Pe lângă proprietatea structurală dată la începutul acestui paragraf, un ansamblu mai are și o proprietate de ordonare care permite execuția rapidă a operațiilor caracteristice: inserare și extragerea celui mai mare (sau a celui mai mic) element. Deoarece dorim să găsim rapid cel mai mare (mic) element, are sens să așezăm cel mai mare (mic) element în rădăcină. Dacă considerăm că și sub-arborii stâng și drept trebuie să fie și ei ansamble, atunci fiecare nod trebuie să fie mai mare (mic) decât descendenții săi.

Aplicând acest demers logic, ajungem la proprietatea de ordonare a ansamblor: valoarea oricărui nod este cel puțin la fel de mare ca valoarea fiilor săi. În **Figura 10.19**, arborele din stânga este un min-ansamblu, în timp ce arborele din dreapta nu este un ansamblu. Pentru ca arborele ansamblu să respecte această proprietate, indiferent de operațiile care se realizează asupra lui, trebuie ca la operațiile de inserare (metoda `insert()`) și ștergere (metoda `deleteMin()`) să se realizeze o reordonare a nodurilor arborelui, astfel încât arborele modificat să respecte și el proprietatea de ordine. Aceasta înseamnă propagarea elementelor înspre și dinspre rădăcină, operație realizată de metoda `fixHeap()`, care filtrează toate elementele șirului (metoda `percolateDown()`).

**Adăugarea unui element într-un ansamblu:** Pentru a fixa ideile, vom conveni ca în continuare prin ansamblu vom înțelege un min-ansamblu. Inserarea unui element  $x$  în ansamblu va decurge astfel: vom crea un nod necompletat în următoarea locație disponibilă (căci altfel arborele nu ar fi complet). Dacă  $x$  poate fi plasat în noul nod fără a încălca proprietatea de ordonare, atunci suntem gata. Altfel, fie  $y$  părintele noului nod; deoarece  $x$  încalcă proprietatea de ordonare, avem evident  $x < y$ . Vom “împinge” pe  $y$  în jos, în locul noului nod inserat, urcând astfel nodul necompletat în sus în arbore. Continuăm acest proces de “împingere” până când noul element  $x$  poate fi plasat în nodul necompletat.

**Figura 10.20** arată că pentru a insera valoarea 14 se creează un nod în urmă-

Figura 10.19: Doi arbori binari compleți. Numai arborele din stânga este un ansamblu (un min-ansamblu). În arborele din dreapta nodul 40 are ca fiu nodul 34, ceea ce încalcă proprietatea de ordonare.

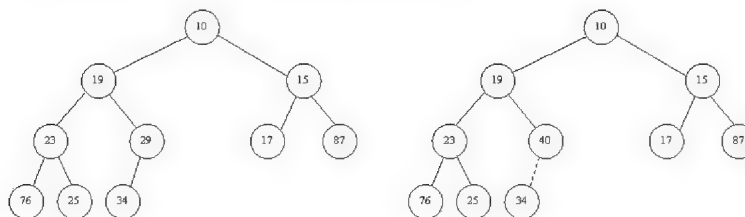
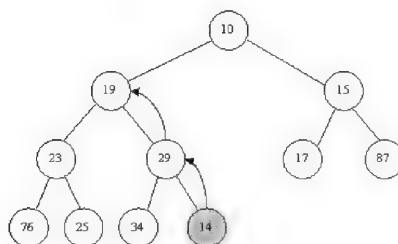


Figura 10.20: Inserarea nodului 14 în ansamblu: se creează o nouă locație, apoi această locație este “împinsă” în sus până ce nodul 14 poate fi așezat în ea. În partea de jos este prezentată reprezentarea sub formă de șir utilizată de metoda `insert()`

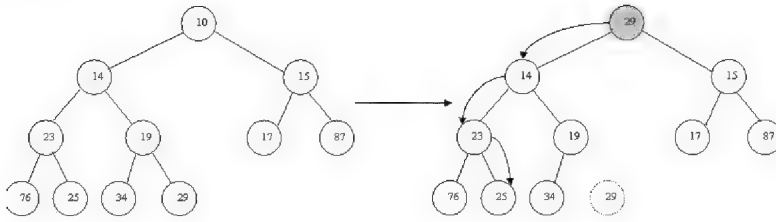


toarea locație disponibilă. Așezarea lui 14 în noul nod ar încălca proprietatea de ansamblu, de aceea 29 “alunecă” în jos în locul noului nod. Pașii sunt continuați până când locul corespunzător pentru 14 este găsit. Denumirea în limba engleză pentru această modalitate de inserare este “percolate up” (filtrare în sus).

Metoda `insert()` din liniile 32-51 (**Listing 10.31**) realizează inserarea obiectului `x` de tip `Comparable` în ansamblu. În liniile 35-39 se verifică dacă tabloul `elements` chiar formează un ansamblu; dacă nu, pur și simplu se adaugă `x` la finalul tabloului și metoda se încheie. Dacă `elements` respectă proprietatea de ordine, atunci se verifică dacă mai este loc în tablou apelând `checkSize()` (linia 41) care dublează numărul de elemente dacă `elements` s-a umplut. Apoi se aplică filtrarea în sus (liniile 43-50), până când `x` ajunge la locul lui în ansamblu.

Timpul necesar pentru a insera un element în ansamblu poate ajunge până la  $O(\log n)$  dacă elementul care trebuie inserat este chiar noul minim, deoarece

Figura 10.21: Crearea unei găuri în locul rădăcinii prin extragerea valorii minime. Din acest moment se caută locul potrivit pentru a insera ultimul nod, 29; se vor urca pe rând nodurile 14, 23, 25, după care se așează 29 în locul lăsat liber de 25



va trebui filtrat în sus până ajunge la rădăcină (iar înălțimea ansamblului este  $\log n$ ). În mod surprinzător, s-a arătat că în medie sunt necesare 2.6 comparații pentru a realiza o inserare, deci în medie un element urcă 1.6 niveluri. Așadar, complexitatea medie a metodei `insert()` este  $O(1)$ .

**Ștergerea unui element dintr-un ansamblu:** Extragerea elementului minim dintr-un ansamblu este realizată foarte asemănător cu inserarea. Găsirea minimumului este ușoară (acesta se află pe poziția 1 în `elements`), partea mai dificilă este extragerea lui. Atunci când minimumul este extras, se creează o “gaură” în locul rădăcinii. Deoarece ansamblul are acum cu un element mai puțin, rezultă că ultimul element,  $x$ , trebuie să urce undeva în sus în ansamblu. Dacă  $x$  poate fi plasat în locul “găurii” din rădăcină, atunci am terminat. Acest lucru este totuși improbabil, de aceea “împingem” gaura în jos în locul fiului mai mic, care urcă în locul găurii. Repetăm acest pas până când  $x$  poate fi plasat în gaură.

**Figura 10.21** arată un ansamblu înainte de ștergerea rădăcinii. După ce nodul 13 a fost extras, trebuie să plasăm nodul 31 undeva în ansamblu; 31 nu poate fi plasat în gaura creată, deoarece ar încălca proprietatea de ordonare. Astfel, vom plasa nodul 14 în locul rădăcinii, și nodul gol alunecă un nivel în jos. Repetăm acest pas din nou, punând pe 19 în gaură și avansând gaura și mai adânc în arbore. Vom pune apoi pe 26 în gaură și creăm o gaură în ultimul nivel. În sfârșit, vom putea pune nodul 31 în gaură. Această strategie este cunoscută sub numele de “filtrare în jos” (percolate down).

Ștergerea unui element din ansamblu este realizată de `deleteMin()` (liniile 145-154). La linia 147 se apelează metoda `findMin()` care întoarce elementul minim, sau aruncă o `UnderflowException` dacă ansamblul este vid (excepția este aruncată mai departe de `deleteMin()`). În linia 149 este

așezat în locul rădăcinii (elementul minim) elementul de pe ultima poziție. Se apelează apoi metoda `percolateDown()` pentru filtra noua rădăcină și a o aduce pe poziția adecvată în ansamblu.

Filtrarea în jos (`percolateDown()`), liniile 113-140) a fost definită separat (spre deosebire de filtrarea în sus, care a fost inclusă în metoda `insert()`) deoarece ea va fi folosită în paragraful următor și la refacerea ansamblului după adăugarea de elemente folosind `append()`. Metoda `percolateDown()` primește ca parametru poziția pe care se află rădăcina (1 în cazul nostru). Ea presupune că atât subarboarele stâng cât și cel drept sunt ansamble și doar rădăcina încalcă proprietatea de ordonare. La linia 116 se salvează rădăcina într-o variabilă temporară. Ciclul `while` de la liniile 118-137 realizează filtrarea propriu-zisă. În liniile 120-126 se calculează care este fiul mai mic al nodului  $i$  (care poate fi  $2*i$  sau  $2*i+1$ ). Dacă rădăcina nu poate fi așezată în locul fiului mai mic (linia 128) atunci fiul mai mic urcă în locul rădăcinii (linia 130), altfel ciclul `while` este întrerupt și se plasează rădăcina pe poziția  $i$  (linia 139). Ciclul `while` se execută până când fie ultimul element (salvat în `tmp`) poate fi așezat în locul nodului curent, fie nodul curent nu mai are fii (și deci putem adăuga rădăcina în locul lăsat liber de nodul curent).

Complexitatea în timp a algoritmului `deleteMin()` este proporțională cu adâncimea arborelui, adică  $O(\log n)$ . De obicei, gaura din rădăcină este coborâtă până în partea de jos a ansamblului, deoarece ultimul element este mare. Având în vedere faptul că adâncimea arborelui este  $\log n$ , rezultă că numărul mediu de execuții ale corpului ciclului `while` este  $O(\log n)$ .

**Construirea unui ansamblu în timp liniar:** Adăugarea metodei `append()` la clasa `BinaryHeap` poate părea nejustificată. De ce să oferim o metodă care strică structura de ansamblu? Răspunsul este că extragerea elementului minim (care folosește de fapt proprietatea de ordonare) dintr-un ansamblu nu este realizată întotdeauna imediat ce am inserat un element. În multe situații practice, se inserează un număr nedeterminat de elemente și abia după aceea se extrage elementul minim. Am văzut în paragrafele anterioare că ștergerea și inserarea în ansamblu se fac în cazul cel mai nefavorabil în  $O(\log n)$ . Astfel, inserarea a  $n$  elemente într-un ansamblu inițial gol s-ar face (în cazul cel mai nefavorabil) în  $\log 1 + \log 2 + \dots + \log n = \log n! \in O(n \log n)$ . Desigur că ținând cont de faptul că metoda `insert()` are totuși o complexitate medie de  $O(1)$ , inserarea celor  $n$  elemente se face într-o complexitate medie de  $O(n)$ . Vom demonstra că folosind  $n$  apeluri ale metodei `append()` urmată de `fixHeap()` putem face același lucru în  $O(n)$ , chiar și în cazul cel mai nefavorabil. Este ușor de văzut că metoda `append()` are complexitate  $O(1)$ , deci  $n$  apeluri ale ei vor fi realizate în  $O(n)$ . Vom arăta imediat că și metoda `fixHeap()` are tot complexitate

$O(n)$ , deci construirea ansamblului s-a realizat în  $O(n)$ .

Înainte de a analiza complexitatea în timp a metodei `fixHeap()`, să vedem cum funcționează ea de fapt. Metoda `fixHeap()` va rearanja elementele tabloului `elements`, astfel încât ele să formeze un ansamblu. `fixHeap()` privește tabloul `elements` ca pe un arbore binar complet și operează de la frunze către rădăcină. Să presupunem că tabloul `elements` conține  $n$  obiecte. Subarborii de rădăcină  $n/2+1, n/2+2, \dots, n$  formează ansamble cu un singur nod, deci nu este necesar să fie modificați. Așadar operarea începe descrescător, de la subarborii de rădăcină  $n/2$  până la cel de rădăcină 1, după cum se vede și din ciclul `for` de la linia 103. Subarborii de rădăcină  $n/2$  poate fi transformat în ansamblu prin aplicarea algoritmului de filtrare în jos, deoarece atât subarborii stâng cât și cel drept sunt deja ansamble. Analog se procedează pentru subarborii de rădăcină  $n/2-1$  până se ajunge la subarborii de rădăcină 1, care este chiar întreg ansamblul.

Să analizăm acum complexitatea în timp a metodei `fixHeap()`. În cazul cel mai nefavorabil, fiecare apel al metodei `percolateDown()` are complexitatea  $h(i)$ , unde  $h(i)$  este înălțimea subarborului de rădăcină  $i$ . Complexitatea metodei `fixHeap()` este așadar proporțională cu suma înălțimilor nodurilor din ansamblu, care este  $N - H - 1 \in O(n)$  în cazul unui ansamblu perfect de înălțime  $H$  (având  $2^{H+1} - 1$  noduri).

Iată și codul complet al clasei `BinaryHeap`:

**Listing 10.31:** Implementarea unei cozi de priorități cu ajutorul unui ansamblu

```
1 package datastructures;
2
3 import exceptions.*;
4
5 /** Implementarea unei cozi de prioritati folosind un
6  * ansamblu
7  */
8 public class BinaryHeap implements PriorityQueue
9 {
10  /** Numarul de elemente din ansamblu. */
11  private int currentSize;
12
13  /** true daca elements formeaza un ansamblu, false altfel */
14  private boolean orderOK;
15
16  /** Elementele din ansamblu. */
17  private Comparable[] elements;
18
```

```

19  /** Capacitatea implicita a ansamblului */
20  private static final int DEFAULT_CAPACITY = 11;
21
22  /** Contruieste un ansamblu, alocand elemente pentru elements
23  * si umpland pozitia 0 cu un element foarte mic */
24  public BinaryHeap(Comparable negInf)
25  {
26      currentSize = 0;
27      orderOK = true;
28      elements = new Comparable[DEFAULT_CAPACITY];
29      elements[0] = negInf;
30  }
31
32  /** Adauga elementul x in ansamblu */
33  public void insert(Comparable x)
34  {
35      if (!orderOK)
36      {
37          append(x);
38          return;
39      }
40
41      checkSize();
42
43      int hole = ++currentSize;
44
45      for ( ; x.compareTo(elements[hole / 2]) < 0; hole /= 2)
46      {
47          elements[hole] = elements[hole / 2];
48      }
49
50      elements[hole] = x;
51  }
52
53  /** Adauga un element ignorand proprietatea de ordonare */
54  public void append(Comparable x)
55  {
56      checkSize();
57      elements[++currentSize] = x;
58
59      if (x.compareTo(elements[currentSize / 2]) < 0)
60      {
61          orderOK = false;
62      }
63  }
64
65  /** Verifica daca sirul elements este plin, si in caz
66  * afirmativ ii dubleaza dimensiunea. */
67  private void checkSize()
68  {

```



```

69     if (currentSize == elements.length - 1)
70     {
71         Comparable[] oldelements = elements;
72         elements = new Comparable[currentSize * 2];
73         for (int i = 0; i < oldelements.length; i++)
74         {
75             elements[i] = oldelements[i];
76         }
77     }
78 }
79
80
81
82 /** Intoarce elementul minim din ansamblu.
83 * @throws UnderflowException Daca ansamblul este gol */
84 public Comparable findMin() throws UnderflowException
85 {
86     if (isEmpty())
87     {
88         throw new UnderflowException("Heap vid");
89     }
90
91     if (!orderOK)
92     {
93         fixHeap();
94     }
95
96     return elements[1];
97 }
98
99 /** Transforma sirul elements intr-un ansamblu filtrand
100 * toate nodurile care nu sunt frunze pana ajunge la radacina */
101 private void fixHeap()
102 {
103     for (int i = currentSize / 2; i > 0; i--)
104     {
105         percolateDown(i);
106     }
107
108     orderOK = true;
109 }
110
111 /** Filtreaza elementul de pe pozitia hole asezandu-l in
112 * subarborele cu radacina hole la locul cuvenit */
113 private void percolateDown(int hole)
114 {
115     int child;
116     Comparable tmp = elements[hole];
117
118     while(hole * 2 <= currentSize)

```

```

119     {
120         child = hole * 2;
121
122         if (child != currentSize &&
123             elements[child + 1].compareTo(elements[child]) < 0)
124         {
125             child++;
126         }
127
128         if (elements[child].compareTo(tmp) < 0)
129         {
130             elements[hole] = elements[child];
131         }
132         else
133         {
134             break;
135         }
136         hole = child
137     }
138
139     elements[hole] = tmp;
140 }
141
142 /** Sterge elementul minim (aflat pe pozitia 1!) si
143 * apeleaza algoritmul de filtrare pentru a refaca ansamblul.
144 * @throws UnderflowException daca ansamblul este vid. */
145 public Comparable deleteMin() throws UnderflowException
146 {
147     Comparable minItem = findMin();
148
149     elements[1] = elements[currentSize--];
150
151     percolateDown(1);
152
153     return minItem;
154 }
155
156 /** true daca ansamblul este vid */
157 public boolean isEmpty()
158 {
159     return currentSize == 0;
160 }
161
162 /** Goleste ansamblul setand dimensiunea la 0 */
163 public void makeEmpty()
164 {
165     currentSize = 0;
166 }
167 }

```

Pentru a testa ansamblul, am creat următorul exemplu simplu:

Listing 10.32: Exemplu de utilizare a unei cozi de prioritate. Programul va afișa:  
Continutul ansamblului: 0 1 2 3 4

```

1 import datastructures.*;
2 import exceptions.*;
3
4 /** Clasa de test simpla pentru cozi de prioritate. */
5 public class TestPriorityQueue
6 {
7     public static void main(String [] args)
8     {
9         PriorityQueue pq = new BinaryHeap(
10             new Integer(Integer.MIN_VALUE));
11
12         pq.insert(new Integer(4));
13         pq.insert(new Integer(2));
14         pq.insert(new Integer(1));
15         pq.insert(new Integer(3));
16         pq.insert(new Integer(0));
17
18         System.out.print("Continutul ansamblului : ");
19         try
20         {
21             for ( ; ; )
22             {
23                 System.out.print(pq.deleteMin() + " ");
24             }
25         }
26         catch (UnderflowException e)
27         {
28         }
29     }
30 }

```

## Rezumat

Am prezentat pe scurt în cadrul acestui capitol cele mai importante structuri de date utilizate de programatori în practică. Fiecare structură de date oferă o interfață clară cu toate operațiile pe care le permite, implementarea ei putând fi făcută în mai multe moduri. Stiva și coada sunt structuri de date elementare, care oferă operații de adăugare respectiv ștergere la un singur capăt în timp constant. Listele înlanțuite permit operații de adăugare și ștergere arbitrare și au avantajul față de șiruri că inserarea unui element nu implică deplasarea elementelor aflate la dreapta. Arborii binari de căutare permit inserarea, căutarea

și ștergerea în timp logaritmice. Ei oferă de asemenea o modalitate de a parcurge în ordine elementele conținute. Tabelele de repartizare permit o implementare extrem de eficientă a operațiilor de inserare și regăsire. În sfârșit, cozile de prioritate permit operații asupra elementului minim (sau maxim) din cadrul structurii în timp logaritmice. Ele au avantajul față de arborii binari (care permit și ei același lucru) că sunt mai simplu de implementat și nu implică reținerea de legături către fii din stânga și dreapta. Structurile de date reprezintă o disciplină de sine stătătoare, iar noi nu am putut aici decât să oferim o introducere în acest domeniu. Cei care doresc să afle mai multe sunt invitați să parcurgă [Tomescu] și [Cormen].

Începând cu următorul capitol, vor fi prezentate metodele fundamentale de elaborare a algoritmilor. Prima dintre ele este metoda căutării cu revenire (back-tracking).

## Noțiuni fundamentale

**ansamblu binar:** o variantă de arbore binar complet în care un nod este fie mai mic fie mai mare decât fii săi.

**arbore:** structură de date formată dintr-o mulțime de noduri și muchii (folosite pentru a lega nodurile între ele) astfel încât nu există noduri care nu sunt legate direct sau indirect și nu se pot realiza circuite.

**arbore binar:** arbore în care fiecare nod are cel mult doi fii.

**arbore binar de căutare:** variantă de arbore binar care suportă eficient operații de adăugare, ștergere și căutare. Fiul din stânga este mai mic decât părintele, iar fiul din dreapta este mai mare.

**clasă iterator:** tip de clasă care permite manipularea unei liste.

**coadă:** structură de date care permite accesul doar la cel mai vechi element adăugat.

**coadă de priorități:** structură de date care permite accesul doar la elementul cel mai mic.

**frunză:** într-un arbore, reprezintă un nod fără fii.

**funcție hash:** funcție care calculează pe baza unui obiect primit, un întreg care va fi asociat cu obiectul respectiv. Doar obiectele care implementează interfața Hashable pot fi folosite de această funcție.

**stivă:** structură de date care permite accesul doar la cel mai recent element inserat.

**structură de date:** o reprezentare a unor date și a operațiilor pe acele date. Prin intermediul structurilor de date se poate obține reutilizarea componentelor.

**tabelă de repartizare (hashtable):** structură de date care permite adăugări, ștergeri și căutări într-un timp mediu constant.

## Erori frecvente

1. Se consideră eroare, încercarea de a accesa sau șterge elemente în cazul în care stiva, coada sau coada de priorități sunt vide.
2. O coadă de priorități nu are nici o legătură cu o coadă. Este doar o coincidență de nume.
3. Funcția `hash()` pentru o tabelă de repartizare returnează o valoare de tip `int`. Este posibil ca în timpul calculelor intermediare să se producă depășiri, așadar este necesar să se verifice ca rezultatul operatorului modulo să fie pozitiv.
4. Performanța unei tabeli de repartizare se degradează semnificativ pe măsură ce factorul de încărcare se apropie de 1.0. Măriți dimensiunea tabeli de îndată ce factorul de încărcare a atins valoarea 0.5.

## Exerciții

### Pe scurt

1. Arătați care sunt rezultatele următoarei secvențe de operații: `adaugă(6)`, `adaugă(12)`, `adaugă(2)`, `adaugă(7)`, `șterge()`, `șterge()` în cazul în care operațiile `adaugă()` și `șterge()` corespund operațiilor de bază pentru:
  - (a) stivă;
  - (b) coadă;
  - (c) arbore binar de căutare;
  - (d) coadă de priorități.
2. Desenați arborii binari de căutare de înălțime 2,3,4,5 și 6 pentru mulțimea de chei {1, 5, 8, 9, 12, 17, 29}!
3. Scrieți ordinea nodurilor rezultată pentru parcurgerea în preordine, inordine și postordine a arborilor de căutare de la exercițiul precedent.
4. Să presupunem că avem numerele de la 1 la 1000 într-un arbore binar de căutare și că dorim să căutăm numărul 363. Care dintre următoarele secvențe nu poate constitui o secvență de noduri examinate?
  - (a) 2, 252, 401, 398, 330, 344, 397, 363;

- (b) 924, 220, 911, 244, 898, 258, 362, 363;
  - (c) 925, 202, 911, 240, 912, 245, 363;
  - (d) 2, 399, 387, 219, 266, 382, 381, 278, 363;
  - (e) 935, 278, 347, 621, 299, 392, 358, 363.
5. Desenați ansamblul care rezultă în urma inserării elementelor 47, 28, 19, 6, 15, 12, 3, 5, 9, 14 într-un ansamblu inițial vid!

### Teorie

1. Să presupunem că am implementa coada din paragraful 10.3 fără a așeza elementele circular. Găsiți o secvență de operații, care repetată la infinit conduce la mărirea nedefinită a tabloului `elements`, chiar dacă în coadă sunt doar câteva elemente.
2. Să presupunem că dorim o structură de date care să suporte *doar* următoarele operații: adăugare, găsirea maximului și ștergerea maximului. Cât de eficient pot fi implementate aceste operații?  
**Indicație:** Se utilizează un ansamblu în care rădăcina este mai mare decât fii.
3. Există o posibilitate de a implementa următoarele operații în timp logaritm în cadrul aceleiași structuri de date: găsire minim și maxim, ștergere minim și maxim, adăugare?  
**Indicație:** Se utilizează un arbore binar de căutare.
4. Există posibilitatea de a implementa următoarele operații în timp constant: `push`, `pop` și găsirea minimului?  
**Indicație:** Se observă că nu se cere și ștergerea minimului. Putem astfel utiliza două stive, una care reține elementele structurii de date, iar alta care reține valorile minime.
5. De ce în cazul metodei `findMin()` din clasa `BinarySearchTree` nu a fost nevoie să salvăm valoarea parametrului `t`, care este modificat de către metodă?
6. Profesorul Știetot are impresia că a descoperit o proprietate remarcabilă a arborilor de căutare. Să presupunem că o căutare pentru cheia  $k$  într-un arbore de căutare termină într-o frunză. Considerăm următoarele mulțimi:  $A$ : nodurile din stânga drumului parcurs până la  $k$ ,  $B$ : mulțimea nodurilor aflate chiar pe acest drum și  $C$ , mulțimea cheilor din dreapta

drumului. Profesorul Știetot susține că toate cheile din A sunt mai mici decât cele din B, care sunt la rândul lor mai mici decât cele din C. Dați cel mai mic contraexemplu care să spulbere afirmația lui Știetot!

7. Care este numărul minim și numărul maxim de elemente într-un ansamblu cu înălțimea  $h$ ?
8. Arătați că un ansamblu cu  $n$  elemente are înălțimea  $\lfloor \log_2 n \rfloor$ !
9. Unde poate să se afle cel mai mare element într-un min-ansamblu?
10. Un tablou cu elementele ordonate crescător este un ansamblu?
11. Secvența 47, 28, 19, 6, 15, 12, 3, 5, 9, 14 este un ansamblu?

### În practică

1. În paragraful 10.2 am văzut cum se poate implementa o stivă folosind un șir pentru a reține elementele. Implementați acum stiva folosind clasa `LinkedList` din paragraful 10.4 și un iterator special, care permite doar operațiile specifice stivei.
2. În paragraful 10.3 am văzut cum se poate implementa o coadă folosind un șir pentru a reține elementele. Implementați acum coada folosind clasa `LinkedList` din paragraful 10.4 și un iterator special, care permite doar operațiile specifice cozii.
3. Scrieți o metodă care afișează elementele dintr-o listă în ordine inversă. Pentru aceasta definiți un iterator care parcurge lista și depune fiecare element într-o stivă. După ce lista a fost parcursă se vor scoate elementele din stivă până când aceasta se golește.

### Proiecte de programare

1. În paragraful 10.6 am văzut cum se implementează o tabelă de repartizare utilizând un tablou pentru a reține elementele. În cazul unui conflict (două elemente care sunt repartizate în același loc) utilizăm o funcție de repartizare pătratică pentru a găsi o poziție liberă în care să inserăm noul element. Există și o altă posibilitate de a gestiona conflictele: fiecare element al tabloului `elements` este de fapt o listă înlănțuită. Astfel, conflictele se vor rezolva așezând elementele succesiv în cadrul listei înlănțuite. Scrieți o clasă `ChainedHashtable` care implementează interfața `Hashtable` folosind metoda descrisă anterior.

## 11. Metoda Backtracking

Ab uno disce omnes (După unul îi  
poți judeca pe toți).

---

Vergiliu, Eneida

Prima metodă de elaborare a algoritmilor pe care o vom prezenta este *backtracking*. Această metodă este utilizabilă pentru un cadru larg de probleme, iar modul ei de aplicare este aproape algoritmic. Dintre toate metodele de elaborare prezentate, *backtracking* este considerată a fi cea mai elementară, deoarece ea se reduce la o parcurgere exhaustivă inteligentă a spațiului de căutare și necesită cele mai puține adaptări ale formei generale pentru a fi aplicată în probleme concrete.

Discuția asupra acestei metode începe cu o prezentare a cadrului în care ea poate fi utilizată și a principiilor esențiale care stau la baza ei. Vom da apoi schema generală de rezolvare a problemelor *backtracking*, urmată de câteva exemple clasice în care această metodă se aplică.

În cadrul acestui capitol vom prezenta:

- Care sunt problemele cărora li se poate aplica metoda *backtracking*;
- Ce sunt condițiile interne și condițiile de continuare, și cum influențează acestea eficiența unui algoritm *backtracking*;
- Care sunt cei patru pași care se aplică stării inițiale a problemei în vederea obținerii soluțiilor;
- Care este schema generală de rezolvare a problemelor *backtracking* și cum se particularizează ea pentru diverse probleme concrete;
- Exemple clasice de probleme care admit rezolvare prin această metodă.



## 11.1 Prezentare generală

În practică apar adeseori situații în care rezolvarea unei probleme se reduce în esență la determinarea unor vectori de forma:

$$x = (x_1, x_2, \dots, x_n)$$

unde:

- fiecare componentă  $x_i$  aparține unei mulțimi finite  $V_i$ ;
- componentele vectorului  $x$  respectă anumite relații, numite *condiții interne*, astfel încât  $x$  este o soluție a problemei dacă și numai dacă aceste condiții sunt satisfăcute de componentele  $x_1, x_2, \dots, x_n$  ale vectorului.

Produsul cartezian  $V_1 \times V_2 \times \dots \times V_n$  se numește *spațiul soluțiilor posibile*. Elementele acestui produs cartezian care respectă condițiile interne se numesc *soluții* ale problemei.

**Exemplul 1:** Fie două mulțimi de litere  $V_1 = \{A, B, C\}$  și  $V_2 = \{M, N\}$ . Se cere să se determine acele perechi  $(x_1, x_2)$  având proprietatea că dacă  $x_1$  este  $A$  sau  $B$ , atunci  $x_2$  nu poate fi  $N$ .

Rezolvarea problemei de mai sus conduce la perechile:

$$(A, M), (B, M), (C, M), (C, N)$$

deoarece din cele șase soluții posibile, doar acestea îndeplinesc condițiile puse în enunțul problemei.

**Exemplul 2:** Se dă mulțimea cu elementele  $\{A, B, C, D\}$ . Se cere să se genereze toate permutările elementelor acestei mulțimi. Se cer așadar cvadrupelele de forma  $x = (x_1, x_2, x_3, x_4)$  care respectă condițiile

- $x_i \neq x_j$  pentru  $i \neq j$ ;
- $x_i$  aparține mulțimii  $V = V_i = \{A, B, C, D\}$ ,  $i = 1, 2, 3, 4$ .

Există mai mulți vectori ce respectă aceste condiții:  $\{A, B, C, D\}$ ,  $\{B, A, C, D\}$ ,  $\{B, C, D, A\}$ ,  $\{B, D, A, C\}$  etc. Mai exact, numărul de permutări ale elementelor unei mulțimi cu 4 elemente este  $4! = 24$ .

O modalitate de rezolvare a problemei ar fi să se genereze toate cele  $4^4 = 256$  elemente ale produsului cartezian  $V_1 \times V_2 \times V_3 \times V_4$  (reprezentând soluțiile

posibile) și să se aleagă dintre ele cele 24 care respectă condițiile interne. Să observăm însă că dacă în loc de 4 elemente mulțimea noastră ar avea 7 elemente, vor exista  $7^7 = 823.543$  variante posibile, dintre care doar  $7! = 5040$  (adică doar 0.6%) vor respecta condițiile interne. Aceasta înseamnă că 99.4% dintre variante sunt generate inutil.

Din cele arătate mai sus reiese că are sens să ne punem problema de a găsi algoritmi mai eficienți, care să evite generarea brutală a întregului spațiu de soluții pentru a selecta apoi soluțiile care respectă condițiile interne. Metoda *backtracking* este o metodă foarte importantă de elaborare a algoritmilor pentru problemele de genul celor descrise mai sus. Deși algoritmi de tip backtracking au și ei, în general, complexitate exponențială, sunt totuși net superiori unui algoritm care generează toate soluțiile posibile.

## 11.2 Prezentarea metodei

Așa cum am văzut în paragraful anterior, *metoda backtracking* urmărește să evite generarea tuturor soluțiilor posibile, reducând astfel drastic timpul de calcul.

Să vedem acum care este modalitatea prin care *backtracking* generează soluțiile, evitând parcurgerea exhaustivă a spațiului de căutare. Componentele vectorului  $x$  primesc valori în ordinea crescătoare a indicilor (vom nota aceste valori cu  $v_1, v_2, \dots, v_n$  cu scopul de a face diferența între o componentă care nu are o valoare atribuită,  $x_k$ , și o componentă care are atribuită o valoare,  $v_k$ ). Aceasta înseamnă că lui  $x_k$  nu i se atribuie o valoare decât după ce  $x_1, x_2, \dots, x_{k-1}$  au primit valori. Mai mult decât atât, valoarea  $v_k$  atribuită lui  $x_k$  va trebui astfel aleasă încât  $v_1, v_2, \dots, v_k$  să respecte anumite condiții, numite *condiții de continuare*, care sunt deduse de către *programator* pe baza *condițiilor interne*. Astfel, dacă în **Exemplul 2**, prima componentă,  $x_1$ , a primit valoarea  $v_1 = A$ , este clar că lui  $x_2$  nu i se va mai putea atribui aceeași valoare (elementele unei permutări trebuie să fie diferite).

Neîndeplinirea condițiilor de continuare exprimă faptul că oricum am alege valorile pentru componentele  $x_{k+1}, \dots, x_n$ , nu vom obține nici o soluție (deci condițiile de continuare sunt strict necesare pentru obținerea unei soluții). Prin urmare, se va trece la atribuirea unei valori lui  $x_k$ , doar dacă condițiile de continuare pentru componentele  $x_1, x_2, \dots, x_k$  (având valorile  $v_1, v_2, \dots, v_k$ ) sunt îndeplinite. În cazul neîndeplinirii condițiilor de continuare, se alege o nouă valoare pentru  $x_k$  sau, în cazul în care mulțimea valorilor posibile,  $V_k$ , a fost epuizată, se încearcă să se facă o nouă alegere pentru componenta *precedentă*,  $x_{k-1}$ , a vectorului, micșorând pe  $k$  cu o unitate. Această revenire la componenta

precedentă dă numele metodei, exprimând faptul că dacă nu putem avansa, urmăm (*engl.* track = "a urmări") înapoi (*engl.* back = "înapoi") secvența curentă din soluție.

Trebuie observat faptul că respectarea condițiilor de continuare de către valorile  $v_1, v_2, \dots, v_k$  nu reprezintă nicidecum o garanție a faptului că vom obține o soluție continuând căutarea cu aceste valori. Deci condițiile de continuare sunt *condiții necesare* pentru ca  $v_1, v_2, \dots, v_k$  să conducă la o soluție, dar nu sunt neapărat *condiții suficiente*.

Alegerea condițiilor de continuare este foarte importantă, o alegere bună ducând la o reducere substanțială a numărului de calcule. În cazul ideal, aceste condiții ar trebui să fie nu numai *necesare*, ci chiar *suficiente* pentru obținerea unei soluții, ceea ce ar garanta că toate secvențele  $v_1, v_2, \dots, v_k$  generate vor conduce în final la o soluție. În practică acest lucru nu este adeseori posibil, de aceea se urmărește găsirea unor condiții de continuare care să fie cât mai "dure", adică să elimine din start cât mai multe soluții neviabile. De obicei, condițiile de continuare reprezintă restricția condițiilor interne la primele  $k$  componente ale vectorului. Evident, condițiile de continuare în cazul  $k=n$  sunt chiar condițiile interne.

De exemplu, o condiție de continuare în cazul problemei permutărilor din **Exemplul 2** ar fi:

$$v_k \neq v_i, \forall i = 1, k-1$$

Prin *metoda backtracking*, orice vector este construit *progresiv*, începând cu prima componentă și mergând către ultima, cu eventuale *reveniri* asupra valorilor atribuite anterior. Reamintim că  $x_1, x_2, \dots, x_n$  primesc valori în mulțimile  $V_1, V_2, \dots, V_n$ . Prin atribuire sau încercări de atribuire eșuate din cauza nerespectării condițiilor de continuare, anumite valori sunt *consumate*. Pentru o componentă oarecare  $x_k$  vom nota prin  $C_k$  mulțimea valorilor consumate la momentul curent. Evident,  $C_k \subset V_k$ .

O descriere completă a stării în care se află un algoritm backtracking la un moment dat se poate face prin precizarea următoarelor elemente:

1. componenta din vector la care s-a ajuns în procesul de căutare, având valoarea  $k$ ;
2. valorile  $v_1, v_2, \dots, v_{k-1}$  care au fost atribuite primelor  $k-1$  componente ale vectorului  $x$ ;
3. mulțimile de valori  $C_1, C_2, \dots, C_k$  care au fost consumate pentru componentele  $v_1, v_2, \dots, v_{k-1}$  și pentru componenta curentă,  $x_k$ .

Această descriere poate fi sintetizată într-un tabel numit *configurație*, având următoarea formă:

$$\left( \begin{array}{c|c} v_1, \dots, v_{k-1} & x_k, x_{k+1}, \dots, x_n \\ C_1, \dots, C_{k-1} & C_k, \phi, \dots, \phi \end{array} \right)$$

Semnificația unei astfel de configurații este următoarea:

1. în încercarea de a construi un vector soluție a problemei, componentelor  $x_1, x_2, \dots, x_{k-1}$  li s-au atribuit valorile  $v_1, v_2, \dots, v_{k-1}$ ;
2. aceste valori satisfac condițiile de continuare;
3. urmează să se atribuiască o valoare componentei  $x_k$ ; deoarece valorile consumate până în prezent sunt cele din mulțimea  $C_k$ , componenta  $x_k$  va putea primi o valoare  $v_k$  din  $V_k - C_k$ ;
4. valorile consumate pentru componentele  $x_1, x_2, \dots, x_k$  sunt cele din mulțimile  $C_1, C_2, \dots, C_k$ , cu precizarea că valorile curente  $v_1, v_2, \dots, v_{k-1}$  sunt consumate, deci apar în mulțimile  $C_1, C_2, \dots, C_{k-1}$ ;
5. pentru componentele  $x_{k+1}, \dots, x_n$  nu s-a încercat nici o atribuire, deci nu s-a consumat nici o valoare și, prin urmare,  $C_{k+1}, \dots, C_n$  sunt vide;
6. până în acest moment au fost construite eventualele soluții de forma:

- $(c_1, \dots)$  cu  $c_1 \in C_1 - \{v_1\}$ ;
- $(v_1, c_2, \dots)$  cu  $c_2 \in C_2 - \{v_2\}$ ;
- .....
- $(v_1, v_2, \dots, v_{k-2}, c_{k-1}, \dots)$  cu  $c_{k-1} \in C_{k-1} - \{v_{k-1}\}$ ;
- $(v_1, v_2, \dots, v_{k-1}, c_k, \dots)$  cu  $c_k \in C_k$ ;

Această ultimă afirmație este mai dificil de înțeles și recomandăm reluarea ei după lecturarea exemplului de mai jos.

În construirea permutărilor mulțimii cu elementele  $\{A, B, C, D\}$  din **Exemplul 2**, pentru  $k = 4$ , configurația

$$\left( \begin{array}{c|c} C & A & B \\ \{A, B, C\} & \{A\} & \{A, B\} \end{array} \middle| \begin{array}{c} x_4 \\ \{A, B\} \end{array} \right)$$

are, conform celor arătate mai înainte, următoarea semnificație:

1. componentele  $x_1, x_2, x_3$  au primit respectiv valorile  $C, A, B$ ;

2. tripletul  $C, A, B$  satisface condițiile de continuare;
3. urmează să se atribuie o valoare componentei  $x_4$ . Componenta  $x_4$  ia valori din mulțimea  $V_4 - C_4$ , adică una din valorile  $\{C, D\}$ ;
4.  $C_1 = \{A, B, C\}, C_2 = \{A\}, C_3 = \{A, B\}, C_4 = \{A, B\}$ ;
5.  $k + 1 = 5 > n$ , deci acest subpunct nu are obiect în această situație;
6. până în acest moment au fost deja construite soluțiile de forma (în ordinea în care au fost descrise la punctul 6 de mai sus):
  - $(A, \dots)$  adică  $(A, B, C, D), (A, B, D, C), (A, C, B, D), (A, C, D, B), (A, D, B, C), (A, D, C, B)$  și
  - $(B, \dots)$  adică  $(B, A, C, D), (B, A, D, C), (B, C, A, D), (B, C, D, A), (B, D, A, C), (B, D, C, A)$ ;
  - soluții de această formă nu există, deoarece  $C_2 - \{v_2\} = \phi$ ;
  - soluții de forma  $(C, A, A, \dots)$ : nu există soluții de această formă;
  - soluții de forma  $(C, A, B, A), (C, A, B, B)$ : nu există soluții de această formă.

*Metoda backtracking* începe a fi aplicată în situația în care nu s-a făcut nici o atribuire asupra componentelor lui  $x$ , deci nu s-a consumat nici o valoare, și se încearcă atribuirea unei valori primei componente. Acest lucru este specificat prin *configurația inițială*, a cărei formă este:

$$\begin{pmatrix} x_1, \dots, x_n \\ \phi, \dots, \phi \end{pmatrix}$$

în care toate mulțimile  $C_k$  sunt vide.

Un alt caz special este cel al *configurațiilor soluție*, având forma:

$$\begin{pmatrix} v_1, \dots, v_n \\ C_1, \dots, C_n \end{pmatrix}$$

cu semnificația că vectorul  $(v_1, \dots, v_n)$  este soluție a problemei. Astfel, pentru

**Exemplul 2**, configurația:

$$\begin{pmatrix} A & B & C & D \\ \{A\} & \{A, B\} & \{A, B, C\} & \{A, B, C, D\} \end{pmatrix}$$

are semnificația că vectorul  $(A, B, C, D)$  constituie o soluție a problemei.

*Metoda backtracking* constă în a porni de la configurația inițială și a-i aplica acesteia una dintre cele patru tipuri de transformări prezentate în continuare, până ce nu se mai poate aplica nici o transformare. Fiecare transformare se aplică în anumite condiții bine precizate: la un moment dat doar o singură transformare poate fi aplicată. Presupunem că ne aflăm în configurația descrisă anterior, în care s-au atribuit valori primelor  $k - 1$  componente. Transformările care pot fi aplicate unei configurații sunt: atribuire și avansează, încercare eșuată, revenire, revenire după construirea unei soluții. Le vom prezenta pe fiecare pe rând.

### 11.2.1 Atribuire și avansează

Acest tip de modificare are loc atunci când mai există valori neconsumate pentru componenta curentă,  $x_k$ , (deci  $C_k \subset V_k$ ), iar valoarea aleasă,  $v_k$ , are proprietatea că  $(v_1, \dots, v_k)$  respectă condițiile de continuare. În acest caz valoarea  $v_k$  se atribuie lui  $x_k$  și se adaugă mulțimii  $C_k$ , după care se avansează la componenta următoare,  $x_{k+1}$ . Această modificare a configurației poate fi reprezentată în felul următor:

$$\left( \begin{array}{c|c} \dots, v_{k-1} & x_k, x_{k+1}, \dots \\ \dots, C_{k-1} & C_k, \phi, \dots \end{array} \right) \xrightarrow{v_k} \left( \begin{array}{c|c} \dots, v_{k-1}, & v_k & x_{k+1}, \dots \\ \dots, C_{k-1}, C_k \cup \{v_k\} & \phi, \dots \end{array} \right)$$

De exemplu, la generarea permutărilor, avem următoarea schimbare de configurație pornind de la starea inițială:

$$\left( \begin{array}{c|c} x_1 & x_2 & x_3 & x_4 \\ \phi & \phi & \phi & \phi \end{array} \right) \xrightarrow{A} \left( \begin{array}{c|c} A & x_2 & x_3 & x_4 \\ \{A\} & \phi & \phi & \phi \end{array} \right)$$

### 11.2.2 Încercare eșuată

Acest tip de modificare are loc atunci când, ca și în cazul anterior, mai există valori neconsumate pentru componenta curentă,  $x_k$ , dar valoarea  $v_k$  aleasă nu respectă condițiile de continuare. În acest caz,  $v_k$  este adăugată mulțimii  $C_k$  (deci este consumată), dar *nu se avansează* la componenta următoare. Modificarea este notată prin:

$$\left( \begin{array}{c|c} \dots, v_{k-1} & x_k, x_{k+1}, \dots \\ \dots, C_{k-1} & C_k, \phi, \dots \end{array} \right) \xrightarrow{v_k} \left( \begin{array}{c|c} \dots, v_{k-1} & x_k, & x_{k+1} \dots \\ \dots, C_{k-1} & C_k \cup \{v_k\}, \phi, \dots \end{array} \right)$$

În exemplul nostru cu generarea permutărilor, următoarea transformare este:

$$\left( \begin{array}{c|ccc} A & x_2 & x_3 & x_4 \\ \hline \{A\} & \phi & \phi & \phi \end{array} \right) \stackrel{A}{\Longleftarrow} \left( \begin{array}{c|ccc} A & x_2 & x_3 & x_4 \\ \hline \{A\} & \{A\}\phi & \phi & \phi \end{array} \right)$$

### 11.2.3 Revenire

Acest tip de transformare apare atunci când toate valorile pentru componenta  $x_k$  au fost consumate ( $C_k = V_k$ ), deci nu mai avem alte posibilități de a da valori acestei componente. În acest caz se revine la componenta precedentă,  $x_{k-1}$ , încercându-se atribuirea unei noi valori acestei componente. Este important de remarcat faptul că revenirea la  $x_{k-1}$  implică faptul că pentru  $x_k$  se vor încerca din nou toate variantele posibile, deci mulțimea  $C_k$  trebuie din nou să fie vidă. Transformarea este notată prin:

$$\left( \begin{array}{c|ccc} \dots, v_{k-1} & x_k, x_{k+1}, \dots \\ \hline \dots, C_{k-1} & C_k, \phi, \dots \end{array} \right) \longleftarrow \left( \begin{array}{c|ccc} \dots, v_{k-2} & x_{k-1}, x_k, \dots \\ \hline \dots, C_{k-2} & C_{k-1}, \phi, \dots \end{array} \right)$$

O situație de *revenire*, în exemplul cu generarea permutărilor este dată de configurația:

$$\left( \begin{array}{ccc|c} 3 & 1 & 2 & x_4 \\ \hline \{1, 2, 3\} & \{1\} & \{1, 2\} & \{1, 2, 3, 4\} \end{array} \right) \longleftarrow \left( \begin{array}{ccc|c} 3 & 1 & & x_3 \ x_4 \\ \hline \{1, 2, 3\} & \{1\} & & \{1, 2\} \phi \end{array} \right)$$

### 11.2.4 Revenire după construirea unei soluții

Acest tip de transformare se realizează atunci când toate componentele vectorului au primit valori care satisfac condițiile interne, adică a fost găsită o soluție. În această situație se revine din nou la cazul în care ultima componentă,  $x_n$ , urmează să primească o valoare.

Transformarea se notează astfel:

$$\left( \begin{array}{c} \dots, v_n \\ \dots, C_n \end{array} \right) \xleftarrow{sol} \left( \begin{array}{c|c} \dots, v_{n-1} & x_n \\ \hline \dots, C_{n-1} & C_n \end{array} \right)$$

În exemplul nostru cu generarea permutărilor, revenirea după găsirea primei soluții este dată de diagrama:

$$\left( \begin{array}{cccc} 1 & 2 & 3 & 4 \\ \{1\} & \{1, 2\} & \{1, 2, 3\} & \{1, 2, 3, 4\} \end{array} \right) \xleftarrow{\text{sol}} \left( \begin{array}{ccc|c} 1 & 2 & 3 & x_4 \\ \{1\} & \{1, 2\} & \{1, 2, 3\} & \{1, 2, 3, 4\} \end{array} \right)$$

Revenirea după construirea unei soluții poate fi considerată ca fiind un caz particular al *revenirii*, dacă adăugăm vectorului soluție  $x$  o componentă suplimentară  $x_{n+1}$ , care nu poate lua nici o valoare ( $V_{n+1} = \phi$ ).

O problemă importantă este cea a încheierii procesului de căutare a soluțiilor, sau, cu alte cuvinte, ne putem pune întrebarea: *transformările succesive aplicate configurației inițiale se încheie vreodată sau continuă la nesfârșit?* Evident că pentru ca *metoda backtracking* să constituie un algoritm trebuie să respecte proprietățile unui algoritm, printre care se află și proprietatea de *finitudine*. Demonstrarea finitudinii algoritmilor de tip backtracking se bazează pe următoarea observație simplă: prin transformările succesive de configurație nu este posibil ca o configurație să se repete, iar numărul de elemente al produsului cartezian  $V_1 \times V_2 \times \dots \times V_n$  este finit. Prin urmare, la un moment dat se va ajunge la configurația:

$$\left( \begin{array}{cccc} x_1 & x_2 & \dots & x_n \\ V_1 & \phi & \dots & \phi \end{array} \right)$$

numită *configurație finală*. În configurația de mai sus ar trebui să aibă loc o revenire (deoarece toate valorile pentru prima componentă au fost consumate), adică o deplasare a barei verticale la stânga. Acest lucru este imposibil, și algoritmul se încheie deoarece nici una din cele patru transformări nu poate fi aplicată. În practică, această încercare de a deplasa bara de pe prima poziție ( $k = 1$ ) pe o poziție anterioară ( $k = 0$ ) este utilizată pe post de condiție de terminare a algoritmului.

Înainte de a trece la implementarea efectivă a *metodei backtracking* în pseudocod, să generăm diagramele de configurație pentru **Exemplul 1**:



$$\begin{aligned}
& \left( \begin{array}{c|c} x_1 & x_2 \\ \phi & \phi \end{array} \right) \xrightarrow{A} \left( \begin{array}{c|c} A & x_2 \\ \{A\} & \phi \end{array} \right) \xrightarrow{M} \left( \begin{array}{c|c} A & M \\ \{A\} & \{M\} \end{array} \right) \xleftarrow{sol} \left( \begin{array}{c|c} A & x_2 \\ \{A\} & \{M\} \end{array} \right) \xleftarrow{N} \\
& \left( \begin{array}{c|c} A & x_2 \\ \{A\} & \{M, N\} \end{array} \right) \xleftarrow{B} \left( \begin{array}{c|c} x_1 & x_2 \\ \{A\} & \phi \end{array} \right) \xrightarrow{B} \left( \begin{array}{c|c} B & x_2 \\ \{A, B\} & \phi \end{array} \right) \xrightarrow{M} \left( \begin{array}{c|c} B & M \\ \{A, B\} & \{M\} \end{array} \right) \\
& \xleftarrow{sol} \left( \begin{array}{c|c} B & x_2 \\ \{A, B\} & \{M\} \end{array} \right) \xleftarrow{N} \left( \begin{array}{c|c} B & x_2 \\ \{A, B\} & \{M, N\} \end{array} \right) \xleftarrow{C} \left( \begin{array}{c|c} x_1 & x_2 \\ \{A, B\} & \phi \end{array} \right) \xrightarrow{C} \\
& \left( \begin{array}{c|c} C & x_2 \\ \{A, B, C\} & \phi \end{array} \right) \xrightarrow{M} \left( \begin{array}{c|c} C & M \\ \{A, B, C\} & \{M\} \end{array} \right) \xleftarrow{sol} \left( \begin{array}{c|c} C & x_2 \\ \{A, B, C\} & \{M\} \end{array} \right) \xrightarrow{N} \\
& \left( \begin{array}{c|c} C & N \\ \{A, B, C\} & \{M, N\} \end{array} \right) \xleftarrow{sol} \left( \begin{array}{c|c} C & x_2 \\ \{A, B, C\} & \{M, N\} \end{array} \right) \xleftarrow{C} \left( \begin{array}{c|c} x_1 & x_2 \\ \{A, B, C\} & \phi \end{array} \right)
\end{aligned}$$

### 11.3 Implementarea metodei backtracking

Procesul de obținere a soluțiilor prin *metoda backtracking* este ușor de programat, deoarece la fiecare pas se modifică foarte puține componente (indicele  $k$ , reprezentând poziția barei, componenta  $x_k$  și mulțimea  $C_k$ ). Algoritmul (scris în pseudocod) de mai jos construiește configurația inițială, după care aplică una dintre cele patru transformări descrise în paragraful anterior, până când se ajunge la configurația finală (adică până când se încearcă o revenire de pe prima poziție):

```

inițializează (citește) mulțimile de valori,  $V_1, \dots, V_n$ 
 $k \leftarrow 1$  //se construiește configurația inițială
pentru  $i = 1, n$ 
     $C_i \leftarrow \phi$ 
//acum începe efectiv aplicarea celor 4 transformări, funcție de caz
cât timp  $k > 0$  //  $k = 0$  înseamnă terminarea căutării
    dacă  $k = n + 1$  atunci //configurația este tip soluție
        reține soluția  $v_1, \dots, v_n$ 
         $k \leftarrow k - 1$  //revenire după soluție
    altfel dacă  $C_k \neq V_k$  atunci //mai există valori neconsumate

```

alege o valoare  $v_k$  din  $C_k - V_k$   
 $C_k = C_k \cup v_k$  //valoarea  $v_k$  este consumată  
 dacă  $v_1, \dots, v_k$  respectă condițiile de continuare atunci  
      $x_k \leftarrow v_k$ ; //atribuie și  
      $k \leftarrow k + 1$ ; //avansează  
 altfel //încercare eșuată, nu fac nimic  
 altfel //revenire  
      $C_k \leftarrow \phi, k \leftarrow k - 1$   
sfârșit cât timp

Algoritmul de mai sus funcționează pentru cazul cel mai general, dar este destul de dificil de programat din cauza lucrului cu mulțimile  $C_k$  și  $V_k$ . Din fericire, adeseori în practică mulțimile  $V_k$  au forma

$$V_k = \{1, 2, \dots, s_k\}$$

deci fiecare mulțime  $V_k$  poate fi reprezentată foarte simplu prin numărul său de elemente,  $s_k$ . Pentru a simplifica și mai mult lucrurile, în cadrul procesului de construire a unei soluții vom alege valorile pentru fiecare componentă  $x_k$  în ordine crescătoare, pornind de la 1 și până la  $s_k$ . În această situație, mulțimea de valori consumate  $C_k$  va fi de forma  $\{1, 2, \dots, v_k\}$  și, drept consecință, va putea fi reprezentată doar prin valoarea  $v_k$ .

Considerațiile de mai sus permit înlocuirea algoritmului anterior, bazat pe mulțimi, cu un algoritm simplificat care lucrează numai cu numere.

Dacă în **Exemplul 1** vom conveni să reprezentăm pe A, B, C prin valorile 1, 2, 3, iar pe M și N prin 1 și 2, configurațiile care se succed în procesul de căutare pot fi reprezentate simplificat astfel:

$$\begin{array}{c}
 \begin{pmatrix} | & x_1 & x_2 \end{pmatrix} \xrightarrow{1} \begin{pmatrix} 1 & | & x_2 \end{pmatrix} \xrightarrow[etc]{1} \begin{pmatrix} 1 & 2 & | \end{pmatrix} \xleftarrow{sol} \begin{pmatrix} 1 & | & x_2 \end{pmatrix}
 \end{array}$$

Particularizarea algoritmului pseudocod prezentat anterior se concretizează în următoarea metodă Java:

Listing 11.1: Metoda backtracking

```

1 public void backtracking ()
2 {
3     int k = 0;
4     while (k >= 0)
5     {

```

118

```

6      if (k == n) //am gasit o solutie
7      {
8          retSol(); //afisam solutia
9          k--; //revenire dupa gasirea unei solutii
10     }
11     else
12     {
13         if (x[k] < s[k]) //mai sunt valori neconsumate
14         {
15             x[k]++; //se ia urmatoarea valoare
16             if (continuate(k)) //respecta cond. de cont?
17             {
18                 k++; //avanseaza
19             }
20         }
21         else
22         {
23             x[k--] = 0; //revenire
24         }
25     }
26 }
27 }

```

Metoda `backtracking()` folosește încă două metode:

- metoda `retSol()`, care, așa cum sugerează și numele ei, reține soluția, constând în valorile vectorului  $x$ . Cel mai adesea această metodă realizează o simplă afișare a soluției și, eventual, o comparare cu soluțiile găsite anterior;
- metoda `continuate(k)` verifică dacă valorile primelor  $k$  componente ale vectorului  $x$  satisfac condițiile de continuare; în cazul afirmativ este returnată valoarea `true`, iar în caz contrar este returnată valoarea `false`.

## 11.4 Probleme clasice rezolvabile prin backtracking

### 11.4.1 Problema generării permutărilor

*Se dă mulțimea  $A$  cu elementele  $\{a_1, a_2, \dots, a_n\}$ . Să se genereze toate permutările elementelor acestei mulțimi.*

Se observă că această problemă este o simplă generalizare a **Exemplului 2** din acest capitol. Mai mult decât atât, problema poate fi redusă la a genera permutările mulțimii de indici  $\{1, 2, \dots, n\}$ . În această situație vom avea

Listing 11.2: Funcția de continuare pentru problema permutărilor

---

```

1 public boolean continuare(int k)
2 {
3     for (int i = 0; i < k; ++i)
4     {
5         if (x[k] == x[i])
6         {
7             return false;
8         }
9     }
10    return true;
11 }

```

---

$V_1 = V_2 = \dots = V_n = \{1, 2, \dots, n\}$ , deci putem aplica varianta simplificată a metodei *backtracking*.

Condițiile interne pe care trebuie să le respecte un vector soluție sunt:

$$x_i \neq x_j \text{ pentru } \forall i, j = 1, n, \quad i \neq j.$$

Condițiile de continuare pentru componenta numărul  $k$  sunt o simplă restricție a condițiilor interne:

$$x_i \neq x_k \text{ pentru } \forall i = 1, k - 1.$$

Codul pentru funcția de continuare este prin urmare foarte simplu, după cum reiese și din **Listing 11.2**.

Metoda *backtracking* pentru generarea permutărilor se obține din metoda *backtracking* pentru cazul general, înlocuind numărul de elemente al mulțimilor  $V_k$  (notat cu  $s_k$ ) prin valoarea  $n$ . Soluția completă a generării permutărilor este prezentată în **Listing 11.4**. Pentru a citi mulțimea de elemente care vor fi permutate am folosit clasa ajutătoare `Reader`, prezentată în volumul întâi, capitolul 4, paragraful *Variabila sistem CLASSPATH*. Reamintim încă o dată codul sursă al acestei clase.

Listing 11.3: Clasa Reader

---

```

1 package io;
2 //clasa va trebui salvata intr-un director cu numele "io"
3 //directorul in care se afla "io" va trebui adaugat
4 //in CLASSPATH
5 import java.io.*;
6 import java.util.StringTokenizer;
7 public class Reader
8 {

```

---

120

```

9  public static String readString()
10 {
11     BufferedReader in = new BufferedReader(
12         new InputStreamReader(System.in));
13     try
14     {
15         return in.readLine();
16     }
17     catch(IOException e)
18     {
19         //ignore
20     }
21     return null;
22 }
23
24 public static int readInt()
25 {
26     return Integer.parseInt(readString());
27 }
28
29 public static double readDouble()
30 {
31     return Double.parseDouble(readString());
32 }
33
34 public static char readChar()
35 {
36     BufferedReader in = new BufferedReader(
37         new InputStreamReader(System.in));
38     try
39     {
40         return (char)in.read();
41     }
42     catch(IOException e)
43     {
44         //ignore
45     }
46     return '\0';
47 }
48
49 public static int[] readIntArray()
50 {
51     String s = readString();
52     StringTokenizer st = new StringTokenizer(s);
53     //aloca memorie pentru sir
54     int[] a = new int[st.countTokens()];
55
56     for (int i = 0; i < a.length; ++i)
57     {
58         a[i] = Integer.parseInt(st.nextToken());

```

```
59     }
60
61     return a;
62 }
63
64 }
```

Prin rularea programului următor se vor genera permutările mulțimii introduse de la tastatură.

---

Listing 11.4: Rezolvarea problemei permutărilor

---

```
1 import io.Reader;
2
3 /**
4  * Program care genereaza permutarile unei multimii
5  * introduse de la tastatura.
6  */
7 public class Permutari
8 {
9     /** Testeaza daca elementul adaugat exista deja.*/
10    public static boolean continuare(int[] x, int k)
11    {
12        for (int i = 0; i < k; ++i)
13        {
14            if (x[k] == x[i])
15            {
16                return false;
17            }
18        }
19
20        return true;
21    }
22
23    /** Construiesc un string care contine solutia curenta.*/
24    public static void retSol(int[] s, int[] x, int nrSol)
25    {
26        System.out.print("Permutarea " + nrSol + ": ");
27        for (int i = 0; i < s.length; i++)
28        {
29            System.out.print(s[x[i] - 1] + " ");
30        }
31        System.out.println();
32    }
33
34    /**
35     * Backtracking standard pentru determinarea
36     * permutarilor multimii.
37     */
38    public static void backtracking(int[] s)
39    {
```

```

40  int k = 0;
41  //aloca memorie pentru sirul de indici
42  int[] x = new int[s.length];
43  int nrSol = 0;
44
45  //initializeaza x
46  for (int i = 0; i < x.length; i++)
47  {
48      x[i] = 0;
49  }
50
51  //procesul de backtracking
52  while (k >= 0)
53  {
54      if (k == x.length) //am gasit o solutie
55      {
56          retSol(s,x,++nrSol) ; //afiseaza solutia
57          k--; //revenire dupa ce o solutie a fost gasita
58      }
59      else
60      {
61          if (x[k] < x.length) //valori neconsumate?
62          {
63              //se ia urmatoarea valoare neconsumata
64              x[k]++;
65
66              //respecta valoarea aleasa
67              //conditia de continuare?
68              if (continuate(x, k))
69              {
70                  k++; //avanseaza
71              }
72          }
73          else
74          {
75              x[k--] = 0; //revenire
76          }
77      }
78  }
79 }
80
81 /** Programul principal.*/
82 public static void main(String[] args)
83 {
84     //citirea elementelor multimii
85     System.out.println("Introduceti elementele multimii " +
86         "(pe o linie, separate prin spatiu):");
87     int[] s = Reader.readIntArray();
88
89     //generarea permutarilor multimii

```

```

90     backtrack ( s );
91 }
92 }

```

### 11.4.2 Generarea aranjamentelor și a combinațiilor

Vom vedea acum cât de simplu se poate adapta algoritmul de generare a permutărilor unei mulțimi pentru a genera aranjamentele și combinațiile acelei mulțimi. Pentru a simplifica lucrurile, vom presupune că mulțimea  $A$  este formată din primele  $n$  numere naturale, adică  $A = \{1, 2, \dots, n\}$ .

Reamintim faptul că prin aranjamente de  $n$  luate câte  $m$  ( $n \geq m$ ), notate  $A_n^m$  se înțeleg toate mulțimile ordonate cu  $m$  elemente formate din elemente ale mulțimii  $A$ , cu alte cuvinte toți vectorii de forma:

$$x = (x_1, \dots, x_m), \text{ unde } x_i \in \{1, 2, \dots, n\}, \quad x_i \neq x_j, \quad \forall i, j = 1, m$$

Se observă că, din punct de vedere al reprezentării formale, singura diferență dintre aranjamente și permutări este că aranjamentele au lungime  $m$  în loc de  $n$ . De altfel, pentru  $m=n$  aranjamentele și permutările coincid.

**Exemplu:** Aranjamentele de 3 luate câte 2 ( $A_3^2$ ) sunt:

$$(1,2), (1,3), (2,1), (2,3), (3,1), (3,2).$$

Condițiile interne și, în consecință, condițiile de continuare sunt identice cu cele de la generarea permutărilor. Prin urmare și funcția de continuare este identică cu cea de la permutări. Unde este totuși diferența? Având în vedere că lungimea vectorului este  $m$  și nu  $n$ , condiția de găsim a unei soluții trebuie adaptată. Prin urmare, în metoda `backtracking()` linia:

```
if (k == n)
```

va fi înlocuită cu:

```
if (k == m)
```

Desigur, aceeași modificare este necesară și în metoda `retSol()`, în care secvența

```
for (int i = 0; i < n; ++i)
```

se va înlocui cu

```
for (int i = 0; i < m; ++i)
```



Listing 11.5: Funcția de continuare pentru combinaări

```

1 public boolean continuare(int k)
2 {
3     if (k > 0 && x[k] <= x[k - 1])
4     {
5         return false;
6     }
7     else
8     {
9         return true;
10    }
11 }

```

Să vedem acum modalitatea de generare a combinațiilor. Reamintim că prin combinații de  $n$  luate câte  $m$  (notat  $C_n^m$ ) se înțeleg toate submulțimile cu  $m$  elemente ale mulțimii  $A = \{1, 2, \dots, n\}$ .

**Exemplu:** Combinațiile de 3 luate câte 2 ( $C_3^2$ ) sunt:

$$(1,2), (1,3), (2,3).$$

Diferența între combinații și aranjamente este dată de faptul că, în cazul combinațiilor, ordinea în care apar componentele nu contează (combinația (1,2) este aceeași cu combinația (2,1) etc). Din acest motiv am optat în exemplul de mai sus să aranjăm componentele unei combinații în ordine crescătoare). Prin urmare, combinațiile unei mulțimii cu  $n$  elemente luate câte  $m$  sunt definite de vectorii:

$$x = (x_1, \dots, x_m), \text{ unde } x_1 < x_2 < \dots < x_m.$$

Condiția de continuare în cazul combinațiilor va fi pur și simplu:

$$x_k > x_{k-1} \text{ pentru } k > 1.$$

Metodele `backtracking()` și `retSol()` sunt, în cazul combinațiilor, identice cu cele de la aranjamente. Diferența apare la funcția de continuare, descrisă în **Listing 11.5**. **Listing 11.6** prezintă varianta mai elegantă a aceleiași funcții.

Problema 3 de la finalul capitolului propune o variantă mai eficientă de generare a combinațiilor în care funcția de continuare este complet eliminată.

Listing 11.6: Funcția de continuare pentru combinări (varianta elegantă)

```

1 public boolean continuare(int k)
2 {
3     return k == 0 || x[k] > x[k - 1];
4 }

```

Figura 11.1: Soluție de așezare a damelor pe tabla de șah

	X		
			X
X			
		X	

### 11.4.3 Problema damelor

*Să se așeze  $n$  dame pe o tablă de șah de dimensiune  $n \times n$  astfel încât damele să nu fie pe aceeași linie, aceeași coloană sau aceeași diagonală (damele să nu se atace între ele).*

Reamintim că în jocul de șah, o damă "atacă" pozițiile aflate pe aceeași linie sau coloană și pe diagonală. O posibilă așezare a damelor pe o tablă de șah de dimensiuni  $4 \times 4$  este dată în **Figura 11.1**.

Să vedem cum putem reformula problema damelor pentru a o aduce la o problemă de tip *backtracking*. Se observă cu ușurință că pe o linie a tablei de șah se poate afla o singură damă, prin urmare putem conveni că prima damă se va așeza pe prima linie, a doua damă pe a doua linie etc. Rezultă că pentru a cunoaște poziția damei numărul  $k$  este suficient să știm coloana pe care aceasta se găsește. O soluție a problemei se poate astfel reprezenta printr-un vector

$$x = (x_1, x_2, \dots, x_n), \quad x_k \in \{1, 2, \dots, n\},$$

unde  $x_k$  reprezintă coloana pe care se găsește dama numărul  $k$ .

Cu această notație, vectorul soluție corespunzător exemplului din **Figura 11.1** este:  $(2, 4, 1, 3)$ .

Să vedem acum care este condiția ca două dame distincte,  $k$  și  $i$ , să se atace:

- în mod cert damele nu pot fi pe aceeași linie;
- damele sunt pe aceeași coloană dacă  $x_k = x_i$ ;

Listing 11.7: Funcția de continuare pentru problema damelor

```

1 public boolean continuare(int k)
2 {
3     for (int i = 0; i < k; ++i)
4     {
5         if (x[i] == x[k] || k-i == Math.abs(x[k]-x[i]))
6         {
7             return false;
8         }
9     }
10    return true;
11 }

```

- damele sunt pe aceeași diagonală dacă se află pe colțurile unui pătrat, adică lungimea ( $|x_k - x_i|$ ) este egală cu lățimea ( $k - i$ ):

$$|x_k - x_i| = |k - i|$$

Condiția de continuare este ca dama curentă,  $k$ , să nu atace nici una dintre damele care deja sunt așezate pe tablă, adică:

$$x_k \neq x_i \text{ și } |x_k - x_i| \neq |k - i| \text{ pentru } \forall i = 1, k - 1.$$

Transpusă în Java, funcția de continuare are forma din **Listing 11.7**.

Modificările care trebuie aduse metodelor `retSol` și `backtracking` sunt minime și le lăsăm ca exercițiu.

*Observație:* Problema damelor este primul exemplu de problemă în care condițiile de continuare sunt necesare, dar nu sunt suficiente. De exemplu (pentru  $n=4$ ), la început, algoritmul va așeza prima damă pe prima coloană, a doua damă pe a treia coloană, iar cea de-a treia damă nu va putea fi așezată pe nici o poziție, fiind necesară o revenire.

#### 11.4.4 Problema colorării hărților

*Se dă o hartă ca cea din figura de mai jos, în care sunt reprezentate schematic 6 țări, dintre care unele au granițe comune. Presupunând că dispunem doar de trei culori (roșu, galben, verde), se cere să se determine toate variantele de colorare a hărții astfel încât oricare două țări vecine (care au frontieră comună) să fie colorate diferit.*

Figura 11.2: Matricea de vecinătăți corespunzătoare hărții din partea stângă

						$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 \end{pmatrix}$
$T_1$	$T_2$		$T_5$			
	$T_3$	$T_4$				
	$T_6$					

Pentru a memora relația de vecinătate între două țări vom utiliza o matrice de dimensiuni  $6 \times 6$ , numită *vecin*, definită prin:

$$vecin[i, j] = \begin{cases} true & \text{dacă țările } T_i \text{ și } T_j \text{ sunt vecine} \\ false & \text{altfel} \end{cases}$$

**Figura 11.2** reprezintă matricea de vecinătăți pentru harta cu 6 țări în care s-a făcut convenția că 1 reprezintă *true* și 0 reprezintă *false*.

Problema se poate generaliza ușor și la o hartă cu  $n$  țări care trebuie colorată cu  $m$  culori. Vom utiliza pentru ușurarea expunerii harta cu 6 țări de mai sus.

În această problemă, un vector soluție  $x = (x_1, x_2, \dots, x_n)$  reprezintă o variantă de colorare a hărții, având semnificația că țara numărul  $i$  va fi colorată cu culoarea  $x_i$ . În exemplul nostru,  $x_i$  poate fi 1, 2 sau 3, corespunzând respectiv culorilor roșu, galben, verde.

Condiția de continuare este ca țara căreia urmărim să îi atribuim o culoare, să aibă o culoare distinctă de țările cu care are graniță. Cu alte cuvinte, trebuie să avem  $x_i \neq x_k$  dacă  $A[i, k] = 1$ ,  $\forall i = 1, k - 1$ .

Funcția de continuare care descrie condiția de mai sus este dată în **Listing 11.8**.

Pentru o mai bună înțelegere a mecanismului *metodei backtracking* aplicată la problema colorării hărților, putem să ne imaginăm că dispunem de 6 cutii identice  $V_1, V_2, \dots, V_6$ , fiecare dintre cutii conținând trei creioane colorate notate cu r - roșu, g - galben, v - verde. Fiecare cutie  $V_k$  conține creioanele care pot fi utilizate pentru colorarea țării  $T_k$ .

O vizualizare a procesului de căutare a soluțiilor poate fi obținută dacă aranjăm cele 6 cutii în ordine (fiecărei țări îi asociem o cutie) și punem un semn înaintea cutiei din care urmează să se aleagă un creion (marcajul corespunde barei verticale de la configurații); inițial acest semn este în stânga primei cutii. Atunci când se alege un creion dintr-o cutie corespunzătoare unei țări el va fi așezat fie pe țara respectivă, dacă nu există o țară vecină cu aceeași culoare,

Listing 11.8: Funcția de continuare pentru problema colorării hărților

```

1 public boolean continuare(int k)
2 {
3     for (int i = 0; i < k; ++i)
4     {
5         if (x[i] == x[k] && vecin[k][i] == 1)
6         {
7             return false;
8         }
9     }
10    return true;
11 }

```

fie lângă cutie în caz contrar. Astfel, mulțimile  $C_i$  de valori consumate la un moment dat sunt alcătuite din creioanele de lângă cutia  $V_i$  și de pe țara  $T_i$ . Cu aceste precizări, cele 4 modificări de configurație au următoarele semnificații concrete:

- *atribuie și avansează*: se așează creionul ales pe țara corespunzătoare și se trece la cutia următoare;
- *încercare eșuată*: creionul ales este așezat lângă cutia din care a fost scos;
- *revenire*: creioanele corespunzătoare țării curente sunt repuse în totalitate la loc în cutie și se trece la cutia precedentă;
- *revenire după găsirea unei soluții*: semnul este adus la stânga ultimei cutii.

Procesul se încheie în momentul în care toate creioanele ajung din nou în cutiile în care se aflau inițial.

## Rezumat

În acest capitol am prezentat *metoda backtracking*, care se reduce în esență la parcurgerea exhaustivă a spațiului de căutare, în care se elimină cu grijă configurațiile care nu pot conduce la o soluție. Metoda *backtracking* se aplică oricărei probleme a cărei soluție se poate scrie sub formă de șir, cu fiecare element al șirului luând valori în cadrul unei mulțimi finite. Elementul esențial care determină eficiența căutării este reprezentat de *condițiile de continuare* care

sunt utilizate pentru a reduce dimensiunile spațiului de căutare. În modelul teoretic al acestei metode, se pornește de la o configurație inițială, căreia i se aplică succesiv anumite transformări până în momentul în care se ajunge la configurația finală. În practică, rezolvarea unei probleme prin această metodă se reduce cel mai adesea la scrierea funcției de continuare (care este o implementare a *condițiilor de continuare*) pentru problema concretă căreia i se aplică.

## Noțiuni fundamentale

**backtracking:** metodă de elaborare a algoritmilor care constă în construirea soluției componentă cu componentă, cu eventuale reveniri asupra componentelor anterioare.

**condiție internă:** condiția pentru ca un șir să reprezinte o soluție a problemei.

**condiție de continuare:** condiție necesară pentru ca un șir parțial să poată conduce la o soluție.

**configurație finală:** configurație căreia nu i se mai poate aplica nici una dintre cele patru transformări și care indică încheierea procesului de căutare.

**configurație inițială:** configurația de la care se pornește în procesul de căutare al soluțiilor.

**diagramă de stare:** diagramă care sintetizează starea în care se află procesul de căutare la un anumit moment.

**soluție:** un șir care respectă condițiile interne.

## Erori frecvente

1. Există adeseori situații în care soluția unei probleme nu se poate scrie direct sub formă de vector, ci trebuie făcute anumite convenții pentru a o reduce la un vector. Așadar, *backtracking* se poate aplica oricărei probleme a cărei soluție se poate *reduce* sau transforma într-un vector. Mulți programatori începători renunță la a încerca să aplice această metodă dacă soluția ei nu este în mod evident structurată sub formă de vector, fără a-și pune problema de a transforma soluția într-o astfel de formă (vezi problema damelor din paragraful 11.4.3, în care soluția se reduce de la o matrice (tabla de șah) la un șir de numere).
2. Deși metoda *backtracking* se poate aplica oricărei probleme a cărei soluție se poate scrie sub formă de șir, aceasta nu înseamnă că *backtracking* este și metoda cea mai eficientă de a o rezolva. De exemplu, și în cazul problemei sortării soluția se scrie sub formă de șir, deci se poate aplica *backtracking*. Totuși aplicarea lui *backtracking* în această situație ar însemna

o generare optimizată a permutărilor elementelor acelui șir, ceea ce ar conduce la o soluție exponențială, deci inutilizabilă. O să prezentăm în capitolul următor metode eficiente de ordonare a unui șir.

3. Adeseori funcția de continuare nu este optim aleasă, în așa fel încât să elimine cât mai multe configurații nefezabile.
4. Se confundă adeseori dimensiunea spațiului de căutare (notată de noi cu  $n$ ) cu dimensiunea individuală a fiecărei mulțimi  $V_k$  (notată de noi cu  $s[k]$ ). Este adevărat ca la multe probleme (de exemplu, permutări) acestea sunt egale, dar la altele (cum ar fi colorarea hărților, combinări, aranjamente), ele diferă.

## Exerciții

### Teorie

1. Enumerați problemele prezentate în cadrul acestui capitol, pentru care condițiile de continuare sunt necesare, dar nu sunt suficiente.
2. Luați o tablă de șah obișnuită și încercați să simulați modul în care se generează soluția problemei damelor pentru  $n = 8$ .
3. Găsiți toate soluțiile de colorare cu trei culori a hărții din **Figura 11.2**.

### În practică

1. Să se afișeze toate modurile în care  $n$  persoane pot fi așezate la o masă rotundă precum și numărul acestora.

*Indicație: Există două posibilități de rezolvare:*

- (a) *Se vor genera toate permutările posibile, prin metoda backtracking, și se vor contoriza. Se va afișa apoi numărul lor. Va trebui însă să Țineți seama de faptul că unele dispuneri sunt identice din cauza mesei circulare;*
- (b) *Mult mai elegant, folosind o observație simplă. Astfel, cu  $n$  obiecte se pot forma  $n!$  permutări. Cum, în cazul dispunerii lor circulare  $1, 2, \dots, n$ , respectiv  $2, 3, \dots, n, 1; \dots; n, 1, 2, \dots, n-1$  sunt identice, rezultă că din  $n$  astfel de permutări trebuie considerată doar cea care începe cu 1. Numărul de permutări va fi așadar  $\frac{n!}{n} = (n-1)!$ .*

2. Idem problema 1, cu precizarea că anumite persoane nu se agreează, deci nu pot fi așezate una lângă cealaltă. La intrare se mai furnizează o matrice simetrică  $A$ , cu următoarea semnificație:

$$A(i, j) = \begin{cases} 1 & \text{dacă nu se agreează} \\ 0 & \text{altfel} \end{cases}$$

3. Să se modifice algoritmul de generare a combinațiilor prezentat în paragraful 11.4.2 astfel încât funcția de continuare să nu mai fie necesară.

*Indicație:* Pentru fiecare componentă  $x[k]$  se pornește cu valoarea  $x[k-1] + 1$ .

4. Se dau  $n$  mulțimi  $A_1, A_2, \dots, A_n$ . Să se afișeze produsul lor cartezian.

*Indicație:* Generarea produsului cartezian înseamnă de fapt generarea întregului spațiu de soluții, adică un backtracking în care funcția de continuare lipsește.

5. Se dă o mulțime  $A = \{1, 2, \dots, n\}$ . Să se afișeze toate submulțimile acestei mulțimi.

*Indicație:* Se generează toți vectorii caracteristici de lungime  $n$ . Prin vector caracteristic se înțelege un vector care are doar valorile 1 sau 0 pentru fiecare element cu semnificația:

$$x[i] = \begin{cases} 1 & \text{dacă } i \text{ aparține submulțimii} \\ 0 & \text{altfel} \end{cases}$$

*Există și o soluție care generează vectorii caracteristici de lungime  $n$  prin adunarea în baza 2. Inițial vectorul este nul, corespunzător mulțimii vide, iar apoi prin adunări repetate se generează toate submulțimile. Atenție, numărul total de submulțimi este  $2^n$ !*

6. O firmă dispune de  $n$  angajați, dintre care  $p$  sunt femei. Firma trebuie să formeze o delegație de  $m$  persoane, dintre care  $k$  sunt femei. Să se afișeze toate delegațiile care se pot forma.

*Indicație:* Pentru a forma o delegație de  $k$  femei din  $p$  disponibile avem la dispoziție  $C_p^k$  variante. Delegația de  $m$  persoane poate fi completată



cu oricare din variantele de  $C_{n-p}^{m-k}$  de alegere a bărbaților din delegație. Așadar numărul total de variante este  $C_p^k * C_{n-p}^{m-k}$ .  
 Generarea efectivă se bazează pe un vector caracteristic cu semnificația:

$$x[i] = \begin{cases} 1 & \text{dacă persoana e femeie} \\ 0 & \text{altfel} \end{cases}$$

Funcția de continuare va număra femeile din delegație și nu va lăsa ca numărul lor să-l depășească pe  $k$ .

7. Se consideră mulțimea  $A = \{1, 2, \dots, n\}$ . Să se furnizeze toate partițiile acestei mulțimi. (O partiție a unei mulțimi este o scriere a mulțimii ca reuniune de submulțimi disjuncte).

*Indicație:* Vom genera partiția sub forma unui vector cu  $n$  componente în care  $x[i] = k$  are semnificația că elementul  $i$  aparține submulțimii  $k$  a partiției considerate. Ca exemplu, pentru  $n = 4$  putem avea, la un moment dat, vectorul  $x = (1, 2, 1, 2)$  ceea ce corespunde partiției:  $A = \{1, 3\} \cup \{2, 4\}$ . Ar fi de remarcat că vectorul caracteristic poate lua valori care vor avea aceeași interpretare, ca de exemplu  $x = (2, 1, 2, 1)$  ceea ce corespunde partiției:  $A = \{2, 4\} \cup \{1, 3\}$ . Dar reuniunea e comutativă și partiția astfel obținută e identică cu cea anterioară. Pentru a evita acest lucru vom impune ca fiecare componentă a vectorului să poată avea cel mult valoarea  $k$ , unde  $k$  este indicele elementului. Semnificația ar fi că elementul cu indicele 1 va putea face parte doar din submulțimea 1, cel cu indicele 2 doar din submulțimile 1 și 2 etc. O altă restricție ar fi aceea că un element nu poate lua o valoare mai mare ca  $\max + 1$ , unde  $\max$  este valoare maximă a elementelor de rang inferior. Acest lucru se justifică prin faptul că  $x = (1, 1, 3, 1)$  nu ar avea nici o semnificație.

8. Un comis-voiajor trebuie să viziteze un număr  $n$  de orașe pornind din orașul numărul 1. El trebuie să viziteze fiecare oraș o singură dată, după care să se întoarcă în orașul 1. Cunoscând legăturile existente între orașe, se cere să se găsească toate rutele posibile pe care le poate efectua comis-voiajorul.

*Indicație:* Se va crea o matrice de adiacență (cunoscută din teoria grafurilor), care este o matrice simetrică:

$$A(i, j) = \begin{cases} 1 & \text{dacă există legătură între orașele } i \text{ și } j \\ 0 & \text{altfel} \end{cases}$$

*Funcția de continuare va testa dacă la elementul actual se poate ajunge din anteriorul, în vectorul  $x$ . Ca observație trebuie spus că pentru a obține soluțiile distincte trebuie făcut un artificiu asemănător cu cel de la problema anterioară.*

9. Idem problema anterioară, cu precizarea că pentru fiecare drum între două orașe se cunoaște distanța care trebuie parcursă. Se cere să se găsească ruta de lungime minimă.

*Indicație: În momentul reținerii soluției se va calcula lungimea drumului parcurs. Se va compara această lungime cu lungimea anterioară considerată minimă și se va reține valoarea actuală minimă împreună cu drumul parcurs.*

*Această problemă este celebră prin faptul că este un exemplu pentru imposibilitatea aflării soluției exacte în timp polinomial. Datorită complexității mari a metodei s-au găsit metode mai puțin complexe (metodele euristice) dar care dau o soluție cu o marjă de aproximare.*

10. Presupunem că avem de plătit o sumă  $s$  și avem la dispoziție un număr nelimitat de bancnote și monede de valoare  $\nu_1, \nu_2, \dots, \nu_n$ . Să se furnizeze toate variantele de plată a sumei utilizând numai aceste monezi.

11. Idem problema anterioară, cu precizarea că trebuie să plătim suma respectivă cu un număr cât mai mic de monede și bancnote.

*Indicație: Față de rezolvarea problemei anterioare se poate face, spre exemplu, o modificare care să compare, în momentul găsirii unei soluții, numărul de monede cu cel găsit la soluțiile anterioare.*

12. Idem problema anterioară pentru cazul în care dispunem doar de un număr  $n_1, n_2, \dots, n_n$  de monede de valoare  $\nu_1, \nu_2, \dots, \nu_n$ .

*Indicație: Deosebirile față de problemele anterioare sunt:*

*\* în vectorul soluție vom reține numărul de monede sau bancnote folosite, nu și valoarea lor. Astfel, fiecărui nivel îi corespunde o anumită valoare;*

\* vom avea grijă ca fiecare componentă a vectorului soluție să nu depășească numărul de monede existent din valoarea care îi corespunde;  
 \* sumele se vor calcula prin cumularea produselor dintre valoare și numărul de valori folosite.

13. Fiind dat un număr natural  $n$ , să se genereze toate partițiile sale. O partiție a unui număr reprezintă scrierea sa ca sumă de numere naturale nenule.
14. Fiind dat un număr natural  $n$ , să se genereze toate descompunerile sale ca sumă de numere prime.

*Indicație: Față de problema anterioară se poate verifica, la continuare, dacă numărul ales este prim.*

15. O fotografie alb-negru este reprezentată sub forma unei matrice cu elemente 0 sau 1. În fotografie sunt reprezentate unul sau mai multe obiecte. Porțiunile corespunzătoare obiectelor au valoarea 1 în matrice. Se cere să se determine dacă fotografia reprezintă unul sau mai multe obiecte.  
*Exemplu: Matricea de mai jos reprezintă două obiecte:*

$$\begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

16. Un teren dreptunghiular este împărțit în  $m \times n$  parcele reprezentate sub forma unei matrice  $A$  cu  $m$  linii și  $n$  coloane. Fiecare element al matricei este un număr real care reprezintă înălțimea parcelei respective. Pe una dintre parcele se află plasată o bilă. Se cere să se furnizeze toate posibilitățile prin care bila poate să părăsească terenul, cunoscut fiind faptul că bila se poate rostogoli numai pe parcele învecinate a căror înălțime este strict inferioară înălțimii parcelei pe care bila se află.

*Indicație: Această și problema anterioară sunt cazuri tipice de backtracking în plan. Ideea rezolvării constă în încercarea de a ajunge la o poziție vecină respectând condițiile problemei. Modalitatea de mișcare este dată de cele 8 direcții cardinale N, NV, V, SV, S, SE, E, NE. Practic, vectorul soluție va conține direcția în care s-a făcut deplasarea.*

17. Pe o tabla de șah de dimensiune  $8 \times 8$  să se poziționeze  $n$  pioni după următoarele reguli:
- (a) Pe fiecare linie se află doi pioni;
  - (b) Pe fiecare coloană se află cel mult doi pioni;
  - (c) Pe fiecare paralelă la diagonala principală se află cel mult doi pioni.

## 12. Divide et impera

Vom găsi o cale, iar dacă nu există,  
vom crea una.

---

Hanibal

În acest capitol vom studia o altă metodă fundamentală de elaborare a algoritmilor, numită *divide et impera*. Ca și *backtracking*, *divide et impera* se bazează pe un principiu extrem de simplu: *descompunem problema în două (sau mai multe) subprobleme de dimensiuni mai mici, rezolvăm subproblemele, iar soluția pentru problema inițială se obține combinând soluțiile subproblemelor în care a fost descompusă*. Rezolvarea subproblemelor se face în același mod cu problema inițială. Procedul se reia până când subproblemele devin atât de simple încât admit o rezolvare imediată.

Încă din descrierea globală a acestei tehnici s-au strecurat elemente de *recursivitate*. Pentru a putea înțelege mai bine această metodă de elaborare a algoritmilor care este eminentă recursivă, vom prezenta pentru început câteva elemente fundamentale referitoare la recursivitate. Continuăm apoi cu prezentarea generală a metodei, urmată de rezolvarea anumitor probleme de *Divide et Impera* deosebit de importante: *căutare binară*, *sortarea prin interclasare*, *sortarea rapidă* și *evaluarea expresiilor aritmetice*.

În cadrul acestui capitol vom prezenta:

- Ce este recursivitatea și care este mecanismul prin care ea funcționează;
- Când se poate aplica metoda *divide et impera* și care este forma ei generală;
- Cum se aplică această metodă pentru a rezolva eficient problema ordonării.

## 12.1 Introducere în recursivitate

În acest paragraf vom reaminti câteva elemente esențiale referitoare la recursivitate. Cei care stăpânesc deja acest mecanism, pot trece direct la prezentarea metodei *Divide et Impera* din paragraful următor.

Având în vedere faptul că recursivitatea este un mecanism de programare general, care nu ține doar de limbajul Java, prezentarea făcută va folosi și limbajul pseudocod la descrierea algoritmilor recursivi, pentru a nu încălca prezentarea cu detalii de implementare.

### 12.1.1 Funcții recursive

*Recursivitatea* este un concept care derivă în mod direct din noțiunea de *recurență matematică*. Recursivitatea este un instrument elegant și puternic pe care programatorii îl au la dispoziție pentru a descrie algoritmi. Este interesant de reținut faptul că programatorii obișnuiau să utilizeze recursivitatea pentru a descrie algoritmi, cu mult înainte ca limbajele de programare să suporte implementarea directă a acestui concept.

Din punct de vedere informatic, o subrutină (procedură<sup>1</sup> sau funcție) recursivă este o *subrutină care se autoapelează*.

Să luăm ca exemplu funcția *factorial*, a cărei definiție matematică recurentă este:

$$fact(n) = \begin{cases} n * fact(n-1) & \text{pentru } n \geq 1 \\ 1 & \text{pentru } n = 0 \end{cases}$$

Din exemplul de mai sus se observă că factorialul este definit funcție de el însuși, dar pentru o valoare a parametrului mai mică cu o unitate. Iată acum care este implementarea recursivă a factorialului, folosind o funcție algoritmică (stânga) și implementarea Java (dreapta):

<pre> funcție fact(n)     dacă n = 0 atunci         fact ← 1     altfel         fact ← n*fact(n-1)     return         </pre>	<pre> public static long fact(int n) {     if (n == 0)         return 1;     else         return n*fact(n-1); }         </pre>
--	--

<sup>1</sup>În algoritmică, prin procedură se înțelege o funcție care nu returnează nici o valoare (de exemplu, în Java o metodă care returnează `void`)

Se observă că funcția de mai sus nu este decât o "traducere" aproape directă a formulei matematice anterioare. Trebuie să remarcăm că, așa cum vom vedea în continuarea acestui paragraf, la baza funcționării acestor funcții stă un mecanism foarte precis, care nu este atât de trivial cum ar părea la prima vedere.

Să luăm ca al doilea exemplu, calculul celebrului *șir al lui Fibonacci*, care este definit recurent astfel:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & \text{pentru } n > 1 \\ n & \text{pentru } n = 0, 1 \end{cases}$$

Implementarea în pseudocod, respectiv Java, a calculului șirului lui Fibonacci este:

<p><u>funcție</u> fib(n)</p> <p><u>dacă</u> n=0 sau n=1 <u>atunci</u></p> <p style="padding-left: 20px;">fib ← n</p> <p><u>altfel</u></p> <p style="padding-left: 20px;">fib ← fib(n-1)+fib(n-2)</p> <p><u>return</u></p>	<p>public static long fib(int n)</p> <p>{</p> <p style="padding-left: 20px;">if (n==0  n==1)</p> <p style="padding-left: 40px;">return n;</p> <p style="padding-left: 20px;">else</p> <p style="padding-left: 40px;">return fib(n-1)+fib(n-2);</p> <p>}</p>
---	---

Se observă că în ambele exemple am început cu așa numita condiție de terminare:

dacă n = 0 sau n = 1 atunci  
fib ← n

care corespunde cazului în care nu se mai fac apeluri recursive. O funcție recursivă care nu are condiție de terminare va genera *apeluri recursive interminabile*, care se soldează în Java cu o eroare de tipul `StackOverflowError` (*depășire de stivă*, deoarece așa cum vom vedea, fiecare apel recursiv presupune salvarea anumitor date pe stivă, iar stiva are o dimensiune finită). Condiția de terminare ne asigură de faptul că atunci când parametrul funcției devine suficient de mic, nu se mai realizează apeluri recursive și funcția este calculată direct.

Ideea fundamentală care stă la baza înțelegerii profunde a mecanismului recursivității este aceea că *în esență, un apel recursiv nu diferă cu nimic de un apel de funcție obișnuit*. Pentru a veni în sprijinul acestei afirmații trebuie să studiem mai în amănunțime ce se petrece în cazul unui apel de funcție.

Se cunoaște faptul că în situația în care compilatorul întâlnește un apel de funcție, acesta predă controlul execuției funcției respective, după care se revine

la următoarea instrucțiune de după apel. Întrebările care apar în mod firesc sunt: *de unde știe compilatorul unde să se întoarcă la terminarea funcției? De unde știe care au fost valorile variabilelor înainte de a se preda controlul funcției?* Răspunsul este simplu: înainte de a realiza un apel de funcție, compilatorul salvează complet starea programului (linia de la care s-a realizat apelul, valorile variabilelor locale, valorile parametrilor de apel) pe stivă, urmând ca la revenirea din subrutină să reîncarce de pe stivă starea care a fost înainte de apel.

Pentru exemplificare să considerăm următoarea procedură (nerecursivă) care afișează o linie a unei matrice. Atât linia care trebuie afișată, cât și matricea sunt transmise ca parametru:

```
procedură AfisLin(a: tmatrice; n, lin: integer)
    pentru i = 1, n
        scrie a[lin, i]
    return
```

Procedura *AfisLin* este apelată de procedura *AfisMat* descrisă mai jos, care afișează, linie cu linie, o întreagă matrice pe care o primește ca parametru:

```
procedură AfisMat(a: tmatrice; n: integer)
    pentru i = 1, n
        AfisLin(a, n, i)
    return
```

Să presupunem că procedura *AfisMat* este apelată într-un program astfel:

```
...
AfisMat(a, 5)
...
```

pentru a afișa o matrice de dimensiuni  $5 \times 5$ .

În momentul în care compilatorul întâlnește acest apel, el salvează pe stivă linia de la care s-a făcut apelul (să spunem 2181), valoarea matricei *a* și alte variabile locale declarate în program:



2181; $AfisMat(a, n); \dots$

Controlul va fi apoi preluat de către procedura  $AfisMat$ , care intră în ciclul *pentru* cu apelul:  $AfisLin(a, n, i)$  aflat, să zicem, la linia 2198.

În acest moment controlul va fi preluat de către procedura  $AfisLin$ , dar nu înainte de a adăuga la vârful stivei linia de la care s-a făcut apelul, valorile parametrilor și a variabilei locale  $i$ :

2198; $AfisLin(n, a, i); i = 1; \dots$
2181; $AfisMat(n, a); \dots$

Procedura  $AfisLin$  va tipări prima linie a matricei, după care execuția ei se încheie. În acest moment compilatorul consultă vârful stivei pentru a vedea unde trebuie să revină și care au fost valorile parametrilor și variabilelor locale înainte de apel. Variabila  $i$  devine 2 și din nou se apelează procedura  $AfisLin$ , etc.

Remarcăm aici faptul că atât procedura  $AfisMat$  cât și procedura  $AfisLin$  utilizează o variabilă locală numită  $i$ . Nu poate exista nici o confuzie între cele două variabile, deoarece în momentul execuției lui  $AfisLin$ , valoarea variabilei  $i$  din  $AfisMat$  este salvată pe stivă.

Să vedem acum evoluția stivei program în cazul calculului recursiv al lui  $fact(5)$ . Presupunem că la linia 2145 are loc apelul recursiv:  $fact \leftarrow n * fact(n - 1)$ .

Pentru a realiza înmulțirea respectivă, trebuie ca întâi să se calculeze  $fact(n - 1)$ . Cum  $n$  are valoarea 5, pe stivă se va depune  $fact(4)$ . Abia după ce valoarea lui  $fact(4)$  va fi calculată se poate calcula valoarea lui  $fact(5)$ . Calculul lui  $fact(4)$  implică însă calculul lui  $fact(3)$ , care implică la rândul lui calculul lui  $fact(2)$ ,  $fact(1)$ ,  $fact(0)$ . Calculul lui  $fact(0)$  se realizează prin atribuire directă, fără nici un apel recursiv:

dacă  $n=0$  atunci

$fact \leftarrow 1$

în acest moment, stiva programului conține toate apelurile recursive realizate până acum:

2145; $fact(0)$ ;
2145; $fact(1)$ ;
2145; $fact(2)$ ;
2145; $fact(3)$ ;
2145; $fact(4)$ ;
xxx; $fact(5)$ ;

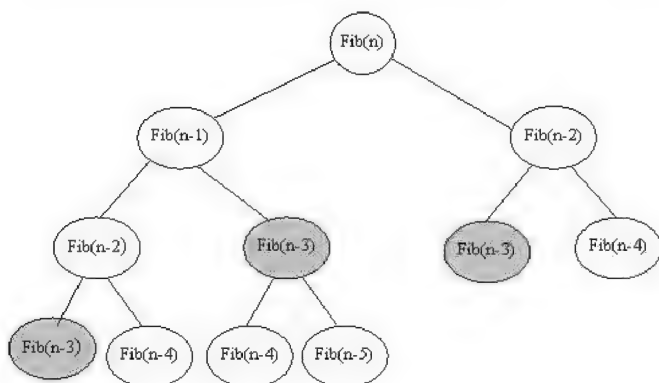
$fact(1)$  fiind calculat, se poate reveni la calculul înmulțirii  $2 * fact(1) = 2$ , apoi,  $fact(2)$  fiind calculat se revine la calculul înmulțirii  $3 * fact(2) = 6$  etc., până se calculează  $5 * fact(4) = 120$  și se revine în programul apelant.

Să vedem acum modul în care se realizează calculul recursiv al șirului lui *Fibonacci*. Vom vedea că timpul de calcul al acestei recurențe este incomparabil mai mare față de calculul factorialului. Să presupunem că funcția  $fib$  se apelează cu parametrul  $n = 3$ . În această situație, se depune pe stivă apelul  $fib(3)$  împreună cu linia de unde s-a realizat apelul (de exemplu, 2160). În linia 2160 a procedurii are loc apelul recursiv:  $fib \leftarrow fib(n-1) + fib(n-2)$  care în cazul nostru,  $n$  fiind 3, presupune calcularea sumei  $fib(2) + fib(1)$ . Această sumă nu poate fi calculată înainte de a-l calcula pe  $fib(2)$ . Calculul lui  $fib(2)$  presupune calcularea sumei  $fib(1) + fib(0)$ .  $fib(1)$  și  $fib(0)$  se calculează direct la următorul apel recursiv, după care se calculează suma lor, rezultând că  $fib(2) = 2$ . Abia acum se revine la suma  $fib(2) + fib(1)$  și se calculează  $fib(1)$ , după care se revine și se calculează  $fib(3)$ .

Modul de calcul al lui  $fib(n)$  recursiv se poate reprezenta foarte sugestiv arborescent. Rădăcina arborelui este  $fib(n)$ , iar cei doi fii sunt apelurile recursive pe care  $fib(n)$  le generează, și anume  $fib(n-1)$  și  $fib(n-2)$ . Apoi se reprezintă apelurile recursive generate de  $fib(n-2)$  ca în **Figura 12.1**.

Din **Figura 12.1** se observă că anumite valori ale șirului lui *Fibonacci* se calculează (inutil) de mai multe ori.  $fib(n)$  și  $fib(n-1)$  se calculează o dată,  $fib(n-2)$  se calculează de două ori,  $fib(n-3)$  de 3 ori etc. Aceasta explică de ce în capitolul 9 am obținut o complexitate exponențială pentru varianta recursivă de calcul a șirului lui Fibonacci.

Figura 12.1: Reprezentarea arborescentă a apelurilor recursive realizate pentru a calcula șirul lui Fibonacci. Se observă că  $\text{Fib}(n-3)$  este calculat de 3 ori.



### 12.1.2 Recursivitatea nu înseamnă recurență

Implementarea recursivă a funcțiilor recurente este ușor de înțeles, datorită mecanismului simplu de transpunere a recurenței într-o funcție recursivă. Totuși, recursivitatea nu se limitează doar la implementarea recurențelor matematice. Putem defini la fel de bine și *operații* recursive. O operație recursivă este definită funcție de ea însăși, dar pentru o problemă de dimensiune cu o unitate mai mică. De exemplu, operația de inversare a  $n$  caractere se poate defini recursiv astfel: se extrage primul caracter din șir, apoi se inversează cele  $n - 1$  caractere rămase după care se adaugă la final caracterul extras. Acest principiu îl aplicăm în exemplul care urmează.

**Exemplu:** Să se scrie o funcție care citește o secvență de caractere până când întâlnește caracterul “.”, după care afișează caracterele în ordine inversă.

Rezolvarea acestei probleme se poate formula recursiv astfel: inversarea caracterelor unui șir de  $n$  elemente implică inversarea caracterelor rămase după citirea primului caracter, și scrierea în final a primului caracter:

$$Inv(n) = Citeste(a) + Inv(n - 1) + Scribe(a)$$

În consecință, metoda de inversare va avea forma (parametrul  $n$  a fost eliminat, el fiind dat în formulă doar pentru claritate):

<p><u>funcție</u> inversare  <u>citește</u> a  <u>dacă</u> a &lt;&gt; '.' <u>atunci</u> inversare  <u>scrie</u> a  <u>return</u></p>	<pre>public static void inversare() {     char a;     a=Reader.readChar();     if (a!='.')     {         inversare();     }     System.out.print(a); }</pre>
--	--

Reamintim că `Reader` este o clasă ajutătoare pentru citirea de date de la tastatură și a fost definită în cadrul primului volum, fiind reluată și în cadrul acestui volum, la paragraful 11.4.1.

Este important de notat că pentru ca metoda să funcționeze corect, variabila *a* trebuie declarată ca variabilă locală; astfel, toate valorile citite vor fi salvate pe stivă, de unde vor fi extrase succesiv (în ordinea inversă citirii) după întâlnirea caracterului ".".

**Exemplu:** Transformarea unui număr din baza 10 într-o bază *b*, mai mică decât 10.

Să ne reamintim algoritmul clasic de trecere din baza 10 în baza *b*. Numărul se împarte la *b* și se reține restul. Câtul se împarte din nou la *b* și se reține restul și se continuă acest procedeu până când câtul devine mai mic decât *b*. Rezultatul se obține prin scrierea în ordine inversă a resturilor obținute.

Formularea recursivă a acestei rezolvări pentru trecerea unui număr *n* în baza *b* este:

$$Transf(n) = \begin{cases} Transf(n \div b) + Scrie(n \bmod b) & \text{pentru } n \geq b \\ - & \text{pentru } n < b \end{cases}$$

Transpunerea recursivă a formulei anterioare este:

<p><u>funcție</u> transform(n:integer)  rest = n mod b  <u>dacă</u> n ≥ b <u>atunci</u> transform(ndivb)  <u>scrie</u> rest  <u>return</u></p>	<pre>public static void transform(int n) {     int rest=n%b;     if(n &gt;= b)     {         transform(n/b);     }     System.out.print(rest); }</pre>
--	--

De remarcat că în această funcție variabila *rest* trebuie să fie declarată local, pentru a fi salvată pe stivă, în timp ce variabila *b* este bine să fie declarată global, deoarece valoarea ei nu se modifică, salvarea ei pe stivă ocupând spațiu inutil.

Odată ce am înțeles mecanismul recursivității, suntem pregătiți pentru a înțelege cea de-a doua metodă de elaborare a algoritmilor, *Divide et Impera*.

## 12.2 Prezentarea metodei Divide et Impera

*Divide et Impera* este o metodă specială prin care se pot aborda anumite categorii de probleme. Ca și celelalte metode de elaborare a algoritmilor, *Divide et Impera* se bazează pe un principiu extrem de simplu: *se descompune problema inițială în două (sau mai multe) subprobleme de dimensiune mai mică, după care soluția problemei inițiale se obține combinând soluțiile subproblemelor în care a fost descompusă*. Procedul de descompunere se repetă până când, după descompuneri succesive, se ajunge la probleme de dimensiune mică, pentru care există rezolvare directă.

Evident, nu orice gen de problemă se pretează la a fi abordată cu *Divide et Impera*. Din descrierea de mai sus reiese că o problemă abordabilă cu această metodă trebuie să aibă două proprietăți:

1. Să se poată descompune în subprobleme;
2. Soluția problemei inițiale să se poată construi simplu pe baza soluției subproblemelor.

Modul în care metoda a fost descrisă, conduce în mod natural la o implementare recursivă, având în vedere faptul că și subproblemele se rezolvă în același mod cu problema inițială. Iată care este forma generală a unei funcții *Divide et Impera*:

```
funcție DivImp (P: Problemă)
    dacă Simplu (P) atunci
        RezolvăDirect (P);
    altfel
        Descompune (P, P1, P2);
        DivImp (P1);
        DivImp (P2);
        Combină (P1, P2);
    return
```

În consecință, putem spune că abordarea *Divide et Impera* implică trei pași la fiecare nivel de recursivitate:

1. *Divide* problema în două subprobleme;
2. *Impera* (*Stăpânește, Cucerește*) cele două subprobleme prin rezolvarea acestora în mod recursiv;
3. *Combină* soluțiile celor două subprobleme în soluția finală pentru problema inițială.

## 12.3 Căutare binară

*Căutarea binară* este o metodă eficientă de regăsire a unor valori într-o secvență ordonată. Căutarea binară este cel mai simplu exemplu de problemă *Divide et Impera*, deoarece în cazul ei se rezolvă doar una din cele două subprobleme, deci faza de recombinație a soluțiilor nu mai este necesară. Enunțul problemei de căutare binară este:

*Se dă un vector cu  $n$  componente (întregi), ordonate crescător și un număr întreg oarecare  $p$ . Să se decidă dacă acest număr se găsește în vectorul dat, și în caz afirmativ să se furnizeze indicele poziției pe care se găsește.*

O rezolvare imediată a problemei presupune parcurgerea secvențială a vectorului dat, până când  $p$  este găsit, sau am ajuns la sfârșitul vectorului. Această rezolvare însă nu folosește faptul că vectorul este sortat.

Căutarea binară procedează în felul următor: se compară  $p$  cu elementul din mijlocul vectorului, dacă  $p$  este egal cu acel element, căutarea s-a încheiat. Dacă este mai mic, se caută doar în prima jumătate, iar dacă este mai mare, se caută doar în a doua jumătate.

Se observă că în această situație problema nu se descompune în două subprobleme care se rezolvă, după care se construiește soluția, ci se reduce la una sau la alta din subprobleme. Cei trei pași ai lui *Divide et Impera* sunt în această situație:

1. *Divide*: împarte șirul de  $n$  elemente în care se realizează căutarea în două șiruri cu  $n/2$  elemente;
2. *Stăpânește*: Caută într-una dintre cele două jumătăți, funcție de valoarea elementului din mijloc;
3. *Combină*: Nu există.

Metoda `binarySearch()` din **Listing 12.1** realizează căutarea elementului `el` în șirul `s`, între indicii `low` și `high`.

**Listing 12.1:** Implementarea căutării binare. Metoda va returna poziția pe care se găsește numărul `el` în șirul `s` sau `-1` dacă `el` nu este găsit.

---

```

1 public static int binarySearch(int[] s, int el, int low,
2                               int high)
3 {
4     if (low <= high) //conditie de oprire
5     {
6         int middle = (low + high) / 2;
7         if (el == s[middle])
8         {
9             return middle;
10        }
11        else
12        {
13            if (el < s[middle])
14            {
15                return binarySearch(s, el, low, middle - 1);
16            }
17            else
18            {
19                return binarySearch(s, el, middle + 1, high);
20            }
21        }
22    }
23    return -1;
24 }
```

Poziția pe care se găsește elementul `el` în șirul `s` este obținută prin apelul:

```
poz = binarySearch(s, el, 0, s.length - 1)
```

**Listing 12.2** prezintă o clasă simplă care utilizează metoda căutării binare pentru a găsi un număr într-un șir citit de la tastatură:

Listing 12.2: Rezolvarea problemei căutării binare

---

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Program ce verifica daca un element introdus
6  * de la tastatura se gaseste in cadrul unui sir ordonat.
7  */
8 public class BinarySearch
9 {
10    /** Metoda de cautare a elementului el in sirul s.*/
11    public static int search(int[] s, int el, int low,
```

```

12         int high)
13     {
14         if (low <= high)
15         {
16             int mid = (low + high) / 2;
17
18             if (el == s[mid])
19             {
20                 //element gasit
21                 return mid;
22             }
23             else
24             {
25                 if (el < s[mid])
26                 {
27                     //cauta in prima jumatate a subsirului
28                     return search(s, el, low, mid - 1);
29                 }
30                 else
31                 {
32                     //cauta in a doua jumatate a subsirului
33                     return search(s, el, mid + 1, high);
34                 }
35             }
36         }
37
38         return -1;
39     }
40
41     /** Programul principal.*/
42     public static void main(String[] args)
43     {
44         //citirea elementelor sirului
45         System.out.println("Introduceti elementele sirului in " +
46             "ordine crescatoare (pe aceeasi linie):");
47         int[] s = Reader.readIntArray();
48
49         //citirea elementului cautat
50         System.out.print("Introduceti elementul cautat: ");
51         int el = Reader.readInt();
52
53         //cautarea elementului
54         int poz = search(s, el, 0, s.length - 1);
55
56         //afisarea rezultatului cautarii
57         if (poz > -1)
58         {
59             System.out.println("Elementul " + el +
60                 " a fost gasit in sir pe pozitia " +
61                 poz);

```



```

62     }
63     else
64     {
65         System.out.println("Elementul " + el +
66             " nu a fost gasit in sir");
67     }
68 }
69 }

```

## 12.4 Sortarea prin interclasare (MergeSort)

Sortarea prin interclasare este, alături de sortarea rapidă (*QuickSort*) și sortarea cu ansamblu (*HeapSort*), una dintre metodele eficiente de ordonare a elementelor unui șir. Enunțul problemei este următorul:

*Să se ordoneze crescător un șir cu  $n$  componente întregi.*

Principiul de rezolvare constă în a împărți șirul care trebuie ordonat în două părți egale și a ordona fiecare jumătate, după care se interclasează cele două jumătăți. Descompunerea în două jumătăți se realizează până când se ajunge la șiruri cu un singur element, care nu mai necesită sortare. Algoritmul de sortare prin interclasare urmează îndeaproape conceptul *Divide et Impera*. Pe scurt, modul lui de operare este următorul:

1. *Divide*: împarte șirul de  $n$  elemente care urmează a fi sortat în două șiruri cu  $n/2$  elemente;
2. *Stăpânește*: Sortează recursiv cele două subșiruri utilizând sortarea prin interclasare;
3. *Combină*: Interclasează subșirurile sortate pentru a obține rezultatul final.

Metoda `mergeSort()` din **Listing 12.3** implementează algoritmul de sortare prin interclasare. Apelul inițial al funcției este:

```
mergeSort(s, 0, s.length - 1);
```

Listing 12.3: Metodă care ordonează șirul `s` folosind sortarea prin interclasare

```

1 public static void mergeSort(int[] s, int low, int high)
2 {
3     if (low < high)
4     {
5         int mid = (low + high) / 2;
6         mergeSort(s, low, mid);
7         mergeSort(s, mid + 1, high);

```

```

8         intercls(s, low, mid, high);
9     }
10 }

```

Metoda de interclasare în acest caz este analoagă cu metoda de interclasare obișnuită a două șiruri, diferența constând în faptul că acum se interclasează două jumătăți ale aceluiași șir, iar rezultatul se va depune în final tot în șirul interclasat. **Listing 12.4** prezintă implementarea completă a sortării prin interclasare, aplicată pe un șir care este preluat de la tastatură.

Listing 12.4: Soluția algoritmului de sortare Mergesort

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Program ce ordonează un șir de numere întregi
6  * folosind metoda MergeSort.
7  */
8 public class MergeSort
9 {
10     /** Metoda de interclasare a celor 2 subsiruri.*/
11     public static void intercls(int low, int mid,
12                                int high, int[] s)
13     {
14         int i = low;
15         int j = mid + 1;
16
17         int[] inter = new int[high + 1];
18         int k = low;
19
20         //interclasarea elementelor
21         while ((i <= mid) && (j <= high))
22         {
23             if (s[i] <= s[j])
24             {
25                 inter[k++] = s[i++];
26             }
27             else
28             {
29                 inter[k++] = s[j++];
30             }
31         }
32
33         //au mai ramas elemente din primul subsir?
34         for (int l = i; l <= mid; l++)
35         {
36             inter[k++] = s[l];
37         }
38

```

```

39  //au mai ramas elemente din cel de-al doilea subsir?
40  for (int l = j; l <= high; l++)
41  {
42      inter[k++] = s[l];
43  }
44
45  //copierea elementelor din sirul inter in sirul s
46  for (i = low; i <= high; i++)
47  {
48      s[i] = inter[i];
49  }
50  }
51
52  /** Metoda de sortare.*/
53  public static void mergeSort(int[] s, int low, int high)
54  {
55      if (low < high) //subsirul are cel putin 2 elemente
56      {
57          int mid = (low + high) / 2;
58          mergeSort(s, low, mid);
59          mergeSort(s, mid + 1, high);
60          intercls(low, mid, high, s);
61      }
62  }
63
64  /** Programul principal.*/
65  public static void main(String[] args)
66  {
67      //citirea elementelor sirului
68      System.out.println("Introduceti elementele sirului " +
69          "(pe aceeasi linie):");
70      int[] s = Reader.readIntArray();
71
72      //sortarea elementelor sirului prin metoda "MergeSort"
73      mergeSort(s, 0, s.length - 1);
74
75      //afisarea rezultatului sortarii
76      System.out.println(
77          "Sirul ordonat prin metoda MergeSort este:");
78      for (int i = 0; i < s.length; i++)
79      {
80          System.out.print(s[i] + " ");
81      }
82  }
83 }

```

## 12.5 Sortarea rapidă (QuickSort)

*Sortarea rapidă* este, așa cum îi spune și numele, cea mai rapidă metodă de sortare prin comparații cunoscută în prezent. Există foarte multe variante ale acestei metode, o parte dintre ele având doar rolul de a micșora timpul de execuție în cazul cel mai nefavorabil. Vom prezenta aici varianta clasică, despre care veți remarca cu surprindere că este neașteptat de simplă. Enunțul problemei este identic cu cel de la sortarea prin interclasare, și anume:

*Să se ordoneze crescător un șir de numere întregi.*

Metoda de sortare rapidă prezentată în acest paragraf este, dintr-un anumit punct de vedere, complementara metodei *Mergesort*. Diferența dintre cele două metode este dată de faptul că, în timp ce la *Mergesort* mai întâi vectorul se împărțea în două părți după care se sorta fiecare parte și apoi se interclasau cele două jumătăți, la *Quicksort* împărțirea se face în așa fel încât cele două șiruri să nu mai necesite a fi interclasate după sortare, adică primul șir să conțină doar elemente mai mici (nu neapărat ordonate) decât elementele celui de-al doilea șir. Rezultă de aici că în cazul lui *Quicksort*, etapa de recombinație este trivială, deoarece problema este astfel împărțită în subprobleme încât să nu mai fie necesară interclasarea șirurilor. Etapele lui *Divide et Impera* pot fi descrise în această situație astfel:

1. *Divide*: Împarte șirul de  $n$  elemente care urmează a fi sortat în două șiruri, astfel încât elementele din primul șir să fie mai mici decât elementele din al doilea șir;
2. *Stăpânește*: Sortează recursiv cele două subșiruri utilizând sortarea rapidă;
3. *Combină*: Șirul sortat este obținut din concatenarea celor două subșiruri sortate.

Funcția care realizează împărțirea în subprobleme (astfel încât elementele primului șir să fie mai mici decât elementele celui de-al doilea) se datorează lui C. A. Hoare, care a găsit o metodă de a realiza această împărțire (numită *partiționare*) în timp liniar.

Metoda de partiționare rearanjează elementele tabloului în funcție de primul element, numit *pivot*, astfel încât elementele mai mici decât primul element sunt trecute în stânga lui, iar elementele mai mari decât primul element sunt trecute în dreapta lui. De exemplu, dacă avem vectorul:

$$a = (7, 8, 5, 2, 3),$$

atunci procedura de partiționare va muta elementele 5, 2 și 3 în stânga lui 7, iar 8 va fi în dreapta lui. Cum se realizează acest lucru? Șirul este parcurs simultan de doi indici: primul indice, *low*, pleacă de la primul element și este incrementat succesiv, iar al doilea indice, *high*, pornește de la ultimul element și este decrementat succesiv. În situația în care  $a[low]$  este mai mare decât  $a[high]$ , elementele se interschimbă. Partiționarea este încheiată în momentul în care cei doi indici se întâlnesc (devin egali) undeva în interiorul șirului. La fiecare pas al algoritmului, fie se incrementează *low*, fie se decrementează *high*; întotdeauna unul dintre cei doi indici, *low* sau *high*, este poziționat pe *pivot*. Atunci când *low* indică pivotul, se decrementează *high*, iar atunci când *high* indică pivotul se incrementează *low*. Iată cum funcționează partiționarea pe exemplul de mai sus. La început, *low* indică primul element, iar *high* indică ultimul element:

$$\begin{array}{cccc} a = (7, 8, 5, 2, 3) \\ \uparrow \quad \quad \uparrow \\ low \quad high \end{array}$$

Deoarece  $a[low] > a[high]$  elementele 7 și 3 se vor interschimba. După interschimbare, pivotul va fi indicat de *high*, deci *low* va fi incrementat:

$$\begin{array}{cccc} a = (3, 8, 5, 2, 7) \\ \uparrow \quad \quad \uparrow \\ low \quad high \end{array}$$

Din nou avem  $a[low] > a[high]$ , elementele 8 și 7 se vor interschimba. După interschimbare, pivotul va fi indicat de *low*, deci *high* va fi decrementat:

$$\begin{array}{cccc} a = (3, 7, 5, 2, 8) \\ \uparrow \quad \quad \uparrow \\ low \quad high \end{array}$$

Din nou avem  $a[low] > a[high]$ , elementele 7 și 2 se vor interschimba. După interschimbare, pivotul va fi indicat de *high*, deci *low* va fi incrementat:

$$\begin{array}{cccc} a = (3, 2, 5, 7, 8) \\ \uparrow \quad \uparrow \\ low \quad high \end{array}$$

De data aceasta avem  $a[low] \leq a[high]$ , deci *low* va fi incrementat din nou, fără a se realiza interschimbări.

Listing 12.5: Metoda de partiționare. Elementele mai mici decât pivotul, `a[low]`, vor fi așezate în stânga lui, iar elementele mai mari în dreapta. Metoda va returna poziția pe care se află pivotul.

```

1 public static int partition(int low, int high)
2 {
3     //variabila care ne spune daca high indica pivotul
4     boolean pozPivot = false;
5     while (low < high) //indicii nu s-au suprapus
6     {
7         if (a[low] > a[high])
8         {
9             interschimba(a, low, high);
10            //celalalt indice indica acum pivotul
11            pozPivot = !pozPivot;
12        }
13        pozPivot ? low++ : high--;
14    }
15
16    return low; //se returneaza pozitia pivotului
17 }

```

În acest moment *low* și *high* s-au suprapus (au devenit egale), deci partiționarea s-a încheiat. Pivotul este pe poziția a 4-a, care este de fapt și poziția lui finală în șirul sortat.

Metoda `partition()` din Listing 12.5 primește ca parametri limitele inferioară, respectiv superioară ale șirului care se partiționează și returnează poziția pe care se află pivotul în finalul partiționării. Poziția pivotului este importantă deoarece ne dă locul în care șirul va fi despărțit în două subșiruri.

Două observații importante merită făcute referitor la metoda `partition()`:

1. Variabila `pozPivot` poate lua valoarea `false` dacă pivotul este indicat de `low`, sau `true` dacă pivotul este indicat de `high`. Atribuirea `pozPivot = !pozPivot` are ca efect schimbarea stării acestei variabile din `false` în `true` sau invers;
2. Metoda se folosește în mod inteligent de transmiterea prin valoare a parametrilor, deoarece modifică variabilele `low` și `high` bazându-se pe faptul că această modificare nu va afecta valorile lui `low` și `high` din metoda `quickSort()`.

Metoda de ordonare propriu-zisă din Listing 12.6 respectă structura *Divide et Impera* obișnuită, doar că funcția de recombinare a soluțiilor nu mai este necesară, deoarece am realizat partiționarea înainte de apel.

Listing 12.6: Metoda de ordonare quickSort

---

```

1 public static void quickSort(int low, int high)
2 {
3     if (low < high) //subsirul mai are cel putin 2 elemente
4     {
5         int mid = partitionare(low, high); //partitioneaza
6         quickSort(low, mid - 1); //sorteaza prima jumatate
7         quickSort(mid + 1, high); //sorteaza a doua jumatate
8     }
9 }

```

Programul din **Listing 12.7** realizează ordonarea folosind *Quicksort* a unui şir citit de la tastatură.

Listing 12.7: Soluția problemei de ordonare prin metoda quicksort

---

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Program ce realizeaza sortarea unui sir de
6  * numere intregi prin metoda quicksort.
7  */
8 public class Quicksort
9 {
10    /** Metoda de partitionare a sirului s.*/
11    public static int partitionare(int[] s, int low, int high)
12    {
13        int pozPivot = 0;
14        int aux;
15
16        while (low < high)
17        {
18            if (s[low] > s[high])
19            {
20                //interschimbarea elementelor s[low] si s[high]
21                aux = s[low];
22                s[low] = s[high];
23                s[high] = aux;
24
25                pozPivot = 1 - pozPivot;
26            }
27
28            if (pozPivot == 0)

```

```

29     {
30         high--;
31     }
32     else
33     {
34         low++;
35     }
36 }
37
38 return low;
39 }
40
41 /** Metoda de sortare a sirului s.*/
42 public static void sort(int[] s, int low, int high)
43 {
44     if (low < high) //subsirul are cel putin 2 elemente
45     {
46         int mid = partitionare(s, low, high);
47         sort(s, low, mid - 1);
48         sort(s, mid + 1, high);
49     }
50 }
51
52 /** Programul principal.*/
53 public static void main(String[] args)
54 {
55     //citirea elementelor sirului
56     System.out.println("Introduceti elementele sirului " +
57         "(pe aceeasi linie):");
58     int[] s = Reader.readIntArray();
59
60     //sortarea elementelor sirului prin metoda "quicksort"
61     sort(s, 0, s.length - 1);
62
63     //afisarea rezultatului sortarii
64     System.out.println(
65         "Sirul ordonat prin metoda quicksort este:");
66     for (int i = 0; i < s.length; i++)
67     {
68         System.out.print(s[i] + " ");
69     }
70 }
71 }

```



## 12.6 Expresii aritmetice

Se dă o expresie aritmetică în care operanzii sunt simbolizați prin litere mici (de la  $a$  la  $z$ ), iar operatorii sunt '+', '-', '/' și '\*' cu semnificația cunoscută. Se cere să se scrie un program care transformă expresia în formă poloneză postfixată.

Reamintim faptul că forma poloneză postfixată (Lukasiewicz) este obținută prin scrierea operatorului după cei doi operanzi, și nu între ei. Această formă are avantajul că nu necesită paranteze pentru a schimba prioritatea operatorilor. Ea este utilizată adeseori în informatică pentru a evalua expresii. Iată câteva exemple:

1.  $a+b$  se scrie  $ab+$
2.  $a*(b+c)$  se scrie  $abc+*$
3.  $(a+b)*(c+d)$  se scrie  $ab+cd+*$

Unul dintre cei mai simpli algoritmi de a trece o expresie în formă poloneză constă în a căuta care este operatorul din expresie cu prioritatea cea mai mică, și de a așeza acest operator la sfârșitul expresiei, urmând ca prima parte a formei poloneze să fie formată din transformarea expresiei din stânga operatorului, iar a doua parte a formei poloneze să fie formată din transformarea expresiei din dreapta operatorului.

Cele două subexpresii urmează a se trata în mod analog, până când se ajunge la o subexpresie de lungime 1, care va fi obligatoriu un operand și care nu mai necesită transformare.

Schematic, dacă avem expresia:

$$E = E1 \text{ op } E2$$

unde  $E1$  și  $E2$  sunt subexpresii, iar  $op$  este operatorul cu prioritatea cea mai mică (deci operatorul unde expresia se poate "rupe" în două), atunci forma poloneză a lui  $E$ , notată  $Pol(E)$ , se obține astfel:

$$Pol(E) = Pol(E1) Pol(E2) op.$$

Expresia de mai sus exprimă faptul că forma poloneză postfixată a lui  $E$  se obține prin scrierea în formă poloneză postfixată a celor două subexpresii, urmate de operatorul care le separă. Expresia de mai sus este o expresie recursivă specifică tehnicii *Divide et Impera*. Etapele sunt în această situație:

1. *Divide*: împarte expresia aritmetică în două subexpresii legate printr-un operator de prioritate minimă;
2. *Stăpânește*: Transformă recursiv în formă poloneză cele două subexpresii;
3. *Combină*: Scrie cele două subexpresii în formă poloneză urmate de operatorul care le leagă.

Soluția problemei expresiilor aritmetice este dată în **Listing 12.8**. Transformarea propriu-zisă este realizată de metoda `polonez()` de la liniile 39-86. `polonez()` primește 3 parametri, care descriu subșirul care trebuie trecut în formă poloneză: `x`, care reprezintă expresia  $E$  la care se adaugă `low` și `high`, care delimitează subșirul din  $E$  care va fi procesat.

Ca orice metodă recursivă, `polonez()` începe la linia 43 cu condiția de terminare:

```
if (low == high)
```

care corespunde cazului în care șirul are un singur caracter (care trebuie în mod obligatoriu să fie un operand) și a cărui transformare în formă poloneză este banală.

Ciclul `while` de la liniile 49-54 realizează un lucru foarte important: elimină eventualele paranteze inutile care înconjoară expresia, pentru a putea găsi ulterior ușor operatorul de unde se “rupe” expresia în două. De exemplu, expresia  $(a + b) * (c + d)$  va fi ruptă într-o primă etapă în  $(a + b)$  și  $(c + d)$ . Ambele expresii rezultate sunt înconjurate de paranteze care sunt acum inutile, și care ar împiedica găsirea operatorului cu prioritate minimă (care trebuie să fie în afara oricărei paranteze). Eliminarea parantezelor exterioare se face utilizând metoda `parant()` de la liniile 14-37, care întoarce `true` dacă parantezele exterioare pot fi eliminate și `false` în caz contrar.

Ciclul `for` de la liniile 56-82 conține pașii de *divide* și *combină* ai metodei. Se încearcă mai întâi găsirea unui operator aditiv (+ sau -) și apoi a unui operator multiplicativ (\* sau /) care să se afle în afara oricărei paranteze. Dacă un astfel de operator este găsit (liniile 73-75), expresia este “ruptă” în acel loc, iar metoda întoarce transformarea în formă poloneză a primei jumătăți, concatenată cu transformarea celei de-a doua jumătăți urmate de operatorul unde s-a realizat împărțirea. Căutarea operatorului de prioritate minimă se face de la dreapta la stânga, și nu de la stânga la dreapta cum era de așteptat, pentru a transforma corect expresii de forma  $a-b-c$  sau  $a/b/c$  (de exemplu, dacă am transforma  $a-b-c$  de la stânga la dreapta, am obține forma poloneză  $-a-bc$ , care se traduce

prin  $a - (b - c)$ , ceea ce este incorect; transformarea corectă, obținută prin parcurgerea de la dreapta la stânga este  $-(b - c)$ . Dacă se ajunge la instrucțiunea `return` de la linia 85, înseamnă că nu s-a găsit nici un operator de la care expresia să se rupă în două, deci expresia este incorectă.

Listing 12.8: Rezolvarea problemei expresiilor aritmetice

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Program care transforma o expresie aritmetica
6  * in forma poloneza postfixata.
7  */
8 public class Expresii
9 {
10     /**
11      * Verifica daca expresia mai este corecta
12      * dupa eliminarea parantezelor exterioare.
13      */
14     public static boolean parant(String x, int low, int high)
15     {
16         int nrp = 0; //numarul de paranteze deschise si neinchise
17
18         for (int i = low + 1; i < high; i++)
19         {
20             if (x.charAt(i) == '(')
21             {
22                 nrp++;
23             }
24
25             if (x.charAt(i) == ')')
26             {
27                 nrp--;
28             }
29
30             if (nrp < 0) //s-a inchis o paranteza fara pereche
31             {
32                 //deci nu putem elimina paranetezele exterioare
33                 return false;
34             }
35
36             return true; //parantezele exterioare pot fi eliminate
37         }
38
39         /** Trece o expresie in forma poloneza postfixata.*/
40         public static String polonez(String x, int low,
41                                     int high, char[][] op)
42         {
43             if (low == high)

```

```

44     {
45         return x.valueOf(x.charAt(low));
46     }
47
48     //elimina parantezele exterioare inutile
49     while (x.charAt(low) == '(' && x.charAt(high) == ')')
50         && parant(x, low, high)
51     {
52         low++;
53         high--;
54     }
55     //cauta locul unde sirul poate fi "rupt" in doua
56     for (int i = 0; i < op.length; i++)
57     {
58         int nrp = 0;
59
60         for (int j = high; j >= low; j--)
61         {
62             if (x.charAt(j) == '(')
63             {
64                 nrp++;
65             }
66
67             if (x.charAt(j) == ')')
68             {
69                 nrp--;
70             }
71             //daca ne aflam in afara parantezelor si am
72             //gasit un operator cu prioritatea adecvata
73             if (nrp == 0 &&
74                 (x.charAt(j) == op[i][0] ||
75                  x.charAt(j) == op[i][1]))
76             {
77                 return polonez(x, low, j - 1, op) +
78                     polonez(x, j + 1, high, op) +
79                     x.valueOf(x.charAt(j));
80             }
81         }
82     }
83     //daca s-a ajuns aici, sirul nu a putut fi "rupt" in
84     //doua, deci expresia este incorecta
85     return "Sirul este incorect";
86 }
87
88 /** Programul principal.*/
89 public static void main(String[] args)
90 {
91     //matricea celor 4 operatori standard
92     char[][] op = { { '+', '-' }, { '*', '/' } };
93

```

```

94 //citirea expresiei aritmetice
95 System.out.print("Introduceti expresia aritmetica: ");
96
97 String x = Reader.readString();
98 if (x.length() == 0) return;
99
100 System.out.println("Forma poloneza postfixata a " +
101     "expresiei aritmetice este: " +
102     polonez(x, 0, x.length() - 1, op));
103 }
104 }

```

## Rezumat

La începutul acestui capitol am prezentat elemente esențiale referitoare la recursivitate. Am văzut care este mecanismul care stă la baza acestei tehnici și faptul că un apel recursiv nu diferă în esență cu nimic de un apel obișnuit. Iată care sunt regulile de bază ale recursivității, pe care este bine să le rețineți și aplicați întotdeauna:

1. *Condiția de terminare:* cel puțin o instanță a problemei trebuie întotdeauna să se poată rezolva fără a utiliza recursivitatea;
2. *Progresul:* orice apel recursiv trebuie să progreseze către condiția de terminare;
3. *Crede și nu cerceta:* întotdeauna trebuie să presupuneți că apelul recursiv funcționează (fără a vă pune problema *cum*);
4. *Suprapunere de apeluri:* evitați să executați același lucru de două ori, prin rezolvarea aceleiași instanțe a problemei în apeluri recursive distince (cum am făcut la șirul lui Fibonacci).

Recursivitatea are multe aplicații concrete, câteva dintre ele fiind prezentate chiar în cadrul acestui capitol.

Introducerea elementelor principale ale recursivității a fost urmată apoi de prezentarea metodei *divide et impera*, precum și a celor trei etape fundamentale care o caracterizează: *divide, cucerește, combină*. Căutarea binară este o metodă de a găsi rapid un element în cadrul unui șir ordonat. *Quicksort* și *Mergesort* sunt metode deosebit de eficiente de a ordona un șir. Problema expresiilor aritmetice presupune trecerea unei expresii din forma standard în forma poloneză postfixată.

## Noțiuni fundamentale

**căutare binară:** metodă de căutare eficientă a unui element în cadrul unui șir ordonat.

**condiție de terminare:** condiție care indică sfârșitul recursiei prin rezolvarea directă a unei instanțe a problemei.

**Mergesort:** algoritm de sortare eficient în care șirul se împarte în două jumătăți, după care se ordonează fiecare jumătate și se combină rezultatul.

**metodă recursivă:** o metodă care se autoapelează direct sau indirect.

**Quicksort:** algoritm de sortare eficient în care șirul se partiționează în două, după care se ordonează fiecare partiție.

**suprapunere de apeluri:** situație în care un algoritm recursiv realizează inutil aceleași apeluri de mai multe ori, scăzând astfel drastic eficiența rezolvării.

## Erori frecvente

1. Cea mai frecventă eroare la începători este de a uita să stabilească o condiție de terminare pentru apelurile recursive.
2. Fiți atenți ca fiecare apel recursiv să constituie un pas către condiția de terminare, altfel recursia este incorectă.
3. Trebuie evitată suprapunerea apelurilor recursive, deoarece ele tind să genereze algoritmi de complexitate exponențială.
4. Complexitatea algoritmilor recursivi trebuie calculată folosind formule de recurență. Nu puteți presupune că un apel recursiv are o complexitate în timp liniară.
5. În cazul unui apel recursiv, doar variabilele locale și parametri actuali se salvează pe stivă. Nu vă bazați pe faptul că variabilele definite în afara funcției sunt salvate la apelul recursiv.
6. Pe de altă parte, evitați să declarați variabile locale sau parametri formali care nu sunt necesari pentru metoda recursivă, pentru a nu umple stiva cu informații inutile.
7. Deși multe probleme admit descompunerea în două sau mai multe sub-probleme, nu întotdeauna soluția problemei inițiale se poate obține pe baza soluției subproblemelor.

## Exerciții

### Teorie

1. Reprezentați evoluția stivei pentru funcțiile recursive din acest capitol.
2. Calculați de câte ori se recalculează valoarea  $F_k$  în cazul calcului recursiv al valorii  $F_n$  a șirului lui Fibonacci, prezentat în paragraful 12.1.1.

### În practică

1. Să se calculeze recursiv și iterativ cel mai mare divizor comun a două numere după formulele (Euclid):

$$\text{cmmdc}(a, b) = \begin{cases} \text{cmmdc}(b, a \bmod b) & \text{pentru } a \bmod b \neq 0 \\ b & \text{altfel} \end{cases}$$

$$\text{cmmdc}(a, b) = \begin{cases} \text{cmmdc}(b, |a - b|) & \text{pentru } a \neq b \\ b & \text{altfel} \end{cases}$$

2. Să se calculeze recursiv și iterativ funcția lui Ackermann, dată de formula:

$$\text{Ack}(m, n) = \begin{cases} n + 1 & \text{pentru } m = 0 \\ \text{Ack}(m - 1, 1) & \text{pentru } n = 0 \\ \text{Ack}(m - 1, \text{Ack}(m, n - 1)) & \text{altfel} \end{cases}$$

3. Să se calculeze combinațiile după formula de recurență din triunghiul lui Pascal:

$$C_n^k = C_{n-1}^k + C_{n-1}^{k-1}$$

Calculați apoi combinațiile după formula clasică:

$$C_n^k = \frac{n!}{k!(n-k)!}$$

Ce constatați? Cum explicați ceea ce ați constatat?

4. Să se calculeze recursiv și iterativ funcția *Manna-Pnueli*, dată de formula:

$$F(x) = \begin{cases} x - 1 & \text{pentru } x \geq 12 \\ F(F(x + 2)) & \text{altfel} \end{cases}$$

5. Să se scrie o funcție care calculează recursiv suma cifrelor unui număr după formula:

$$S(n) = \begin{cases} n \bmod 10 + S(n \div 10) & \text{pentru } n > 0 \\ 0 & \text{pentru } n = 0 \end{cases}$$

6. Se consideră două șiruri definite recurent după formulele:  
 $a_n = \frac{a_{n-1} + b_{n-1}}{2}$  și  $b_n = \sqrt{a_{n-1} b_{n-1}}$ , cu  $a_0 = a$  și  $b_0 = b$ .  
 Să se scrie un program recursiv care calculează aceste șiruri.

7. (*Partițiile unui număr*) Un număr natural  $n$  se poate descompune ca sumă descrescătoare de numere naturale. De exemplu, pentru numărul 4 avem descompunerile  $2+1+1$  sau  $3+1$ . Prin  $P(n, k)$  se notează numărul de posibilități de a-l descompune pe  $n$  ca sumă (descrescătoare) de  $k$  numere. De exemplu,  $P(4, 2) = 2$  ( $4 = 3 + 1, 4 = 2 + 2$ ). Numerele  $P(n, k)$  verifică relația de recurență:

$$P(n + k, k) = P(n, 1) + P(n, 2) + \dots + P(n, k)$$

cu  $P(n, 1) = P(n, n) = 1$ .

Să se calculeze numărul total de descompuneri ale numărului  $n$ .

8. Să se modifice metoda `polonez()` din **Listing 12.8** astfel încât să poată transforma corect și expresii în care operanzii au lungimi mai mari de un caracter. De exemplu  $(max + 1) * (min + 2)$  etc.
9. Să se scrie o funcție care calculează maximul elementelor unui șir utilizând tehnica Divide et Impera.

*Indicație: Se împarte șirul în două jumătăți egale, se calculează recursiv maximul celor două jumătăți și se alege numărul mai mare.*

10. (*Turnurile din Hanoi*) Se dau trei tije simbolizate prin literele A, B și C. Pe tija A se află  $n$  discuri de diametre diferite așezate descrescător în ordinea diametrelor, cu diametrul maxim la bază. Se cere să se mute discurile pe tija B respectând următoarele reguli:



- (a) la fiecare pas se mută un singur disc;
- (b) nu este permisă așezarea unui disc cu diametru mai mare peste un disc cu diametrul mai mic.

*Indicație: Formularea recursivă a soluției este: se mută primele  $n-1$  discuri de pe tija A pe tija C folosind ca tijă intermediară tija B; se mută discul rămas pe A pe tija B; se mută discurile de pe tija C pe tija B folosind ca tijă intermediară tija A. Parcurgerea celor trei etape permite definirea recursivă a șirului  $H(n, a, b, c)$  astfel:*

$$H(n, a, b, c) = \begin{cases} ab & \text{dacă } n = 1 \\ H(n-1, a, c, b), ab, H(n-1, c, b, a) & \text{dacă } n > 1 \end{cases}$$

*Exemplu: Pentru  $n = 2$  avem:*

$$H(2, a, b, c) = H(1, a, c, b), ab, H(1, c, b, a) = ac, ab, cb$$

11. Scrieți un program în care calculatorul să ghicească cât se poate de repede un număr natural la care v-ați gândit. Numărul este cuprins între 1 și 32.000. Atunci când calculatorul propune un număr  $i$  se va răspunde prin 1, dacă numărul este prea mare, 2 dacă numărul este prea mic și 0 dacă numărul a fost ghicit.

*Indicație: Problema folosește metoda căutării binare prezentată în acest capitol.*

12. (*Problema tăieturilor*) Se dă o bucată dreptunghiulară de tablă de dimensiune  $l \times h$ , având pe suprafața ei  $n$  găuri de coordonate numere întregi (colțul din stânga jos al tablei este considerat centrul sistemului de coordonate). Să se determine care este bucata de arie maximă fără găuri care poate fi decupată din suprafața originală. Sunt permise doar tăieturi orizontale sau verticale.

*Indicație: Se caută în bucata curentă prima gaură. Dacă o astfel de gaură există, atunci problema se împarte în alte patru subprobleme de același tip. Dacă suprafața nu are nici o gaură, atunci se compară suprafața ei cu suprafețele fără gaură obținute până la acel moment. Dacă suprafața este mai mare, atunci se rețin coordonatele ei.*

Coordonatele găurilor sunt date în doi vectori  $xv$  și  $yv$ . Coordonatele dreptunghiurilor care apar pe parcursul problemei sunt reținute prin colțul stânga jos  $(x,y)$ , lungime și lățime  $(l,h)$ .

Pentru a se afla în interiorul unui dreptunghi o gaură trebuie să îndeplinească simultan condițiile:

- (a)  $xv(i) > x$ ;
- (b)  $xv(i) < x + l$ ;
- (c)  $yv(i) > y$ ;
- (d)  $yv(i) < y + h$ .

Tăietura verticală prin această gaură determină două dreptunghiuri:

- (a)  $x, y, xv(i) - x, h$ ;
- (b)  $xv(i), y, l + x - xv(i), h$ .

Tăietura orizontală prin această gaură determină alte două dreptunghiuri:

- (a)  $x, y, l, yv(i) - y$ ;
- (b)  $x, yv(i), l, h + y - yv(i)$ .

## 13. Algoritmi Greedy

Am constatat că cu cât muncesc  
mai mult, cu atât am mai mult  
noroc.

---

Thomas Jefferson

Algoritmii aplicați problemelor de optimizare (în care se urmărește obținerea minimului sau maximum unei funcții obiectiv) sunt, în general, compuși dintr-o secvență de pași, la fiecare pas existând mai multe alegeri posibile. Pentru multe probleme de optimizare, utilizarea metodei programării dinamice (prezentată în capitolul 14) în vederea determinării celei mai bune soluții se dovedește a fi o strategie prea complicată. Un algoritm Greedy va alege la fiecare moment soluția care pare a fi cea mai bună la momentul respectiv. Este vorba deci despre o alegere optimă, făcută local, cu speranța că ea va conduce la un optim global. Acest capitol tratează probleme de optimizare care pot fi rezolvate cu ajutorul algoritmilor Greedy. Algoritmii Greedy conduc în multe cazuri la soluții optime, dar nu întotdeauna... În secțiunea 13.1 vom prezenta mai întâi o problemă simplă dar netrivială, problema selectării activităților, a cărei soluție poate fi calculată în mod eficient cu ajutorul unei metode de tip Greedy. Mai departe, în secțiunea 13.2 se recapitulează câteva elemente de bază ale metodei Greedy. Capitolul se încheie cu prezentarea câtorva probleme specifice.

Metoda Greedy este destul de puternică și este aplicată cu succes unui spectru larg de probleme. Lucrările despre teoria grafurilor conțin mai mulți algoritmi care pot fi priviți ca aplicații ale metodei Greedy, cum ar fi algoritmii de determinare a arborelui parțial de cost minim (Kruskal, Prim) sau algoritmul lui Dijkstra pentru determinarea celor mai scurte drumuri pornind dintr-un vârf.

În cadrul acestui capitol vom vedea:

- Care sunt elementele care caracterizează o strategie greedy;

- Ce este substructura optimă (principiul optimalității) și principiul alegerii greedy;
- Cum se poate demonstra corectitudinea unui algoritm.

### 13.1 Problema spectacolelor (selectarea activităților)

Primul exemplu pe care îl vom considera este o problemă de repartizare a unei resurse (o sală de spectacol) mai multor activități care concurează pentru a obține resursa respectivă (diferite spectacole care au loc în sala respectivă). Vom vedea că un algoritm de tip Greedy reprezintă o metodă simplă și elegantă pentru programarea unui număr maxim de spectacole care nu se suprapun (numite activități compatibile reciproc).

*Să presupunem că dispunem de o mulțime  $S = 1, 2, \dots, n$  de  $n$  activități (spectacole) care doresc să folosească o aceeași resursă (sala de spectacole). Această resursă poate fi folosită de o singură activitate la un moment dat. Fiecare activitate  $i$  are un timp de start  $s_i$  și un timp de terminare  $t_i$ , unde  $s_i \leq t_i$ . Dacă este selectată activitatea  $i$ , ea se desfășoară pe durata intervalului  $[s_i, t_i)$ . Două activități sunt compatibile dacă duratele lor de desfășurare sunt disjuncte. Problema spectacolelor (selectării activităților) constă în selectarea unei mulțimi maxime de activități compatibile între ele.*

Un algoritm Greedy pentru această problemă este descris de următoarea funcție, prezentată în pseudocod. Vom presupune că spectacolele (adică datele de intrare) sunt ordonate crescător după *timpul de terminare*:

$$t_1 \leq t_2 \leq \dots \leq t_n.$$

În cazul în care activitățile nu sunt ordonate astfel, ordonarea poate fi făcută în timpul  $O(n * \log n)$  (folosind *Mergesort* sau *Quicksort* prezentate în capitolul anterior). Algoritmul de mai jos presupune că datele de intrare  $s$  și  $t$  sunt reprezentate ca șiruri.

```
funcție SELECT-SPECTACOLE-GREEDY( $s, t$ )
//SS = mulțimea spectacolelor selectate
SS ← {1}
//uss = indicele Ultimului Spectacol
//Selectat
uss ← 1
```

```

    pentru  $sc = 2, n$  //  $sc$  este spectacolul curent
        dacă  $s_{sc} \geq t_{uss}$  atunci
            //spectacolul curent începe
            //după ce  $uss$  s-a terminat, deci
            //se adaugă  $sc$  la spect. selectate
             $SS \leftarrow SS \cup \{sc\}$ 
            //ultimul spectacol selectat devine  $sc$ 
             $uss \leftarrow sc$ 
    return  $SS$ 

```

În mulțimea  $SS$  se introduc spectacolele care au fost selectate. Variabila  $uss$  identifică ultimul spectacol introdus în  $SS$ . Deoarece activitățile sunt considerate în ordinea crescătoare a timpilor lor de terminare,  $t_{uss}$  va reprezenta întotdeauna timpul maxim de terminare a oricărei activități din  $SS$ . Aceasta înseamnă că:

$$t_{uss} = \max\{t_k | k \in SS\}$$

La începutul algoritmului, mulțimea spectacolelor selectate,  $SS$ , se inițializează cu activitatea 1 (deoarece această activitate se termină cel mai repede), iar variabila  $uss$  (ultimul spectacol selectat) ia ca valoare această activitate. În continuare, în ciclul *pentru* se consideră pe rând fiecare activitate,  $sc$ . Aceasta se adaugă mulțimii  $SS$  dacă este compatibilă cu celelalte activități deja selectate. Pentru a vedea dacă spectacolul curent,  $sc$ , este compatibil cu toate celelalte activități existente la momentul curent în  $SS$ , este suficient ca momentul de start  $s_{sc}$  să nu fie mai devreme decât momentul de terminare  $t_{uss}$  al activității cel mai recent adăugate mulțimii  $SS$ . Dacă spectacolul curent,  $sc$ , este compatibil, atunci el este adăugat mulțimii  $SS$ , iar variabila  $uss$  este actualizată. Funcția *SELECT-SPECTACOLE-GREEDY* este foarte eficientă. Ea poate planifica o mulțime  $S$  de  $n$  activități în  $O(n)$ , presupunând că activitățile au fost deja ordonate după timpul lor de terminare. Activitatea aleasă de *SELECT-SPECTACOLE-GREEDY* este întotdeauna cea cu primul timp de terminare care poate fi planificată fără suprapuneri.

### 13.1.1 Demonstrarea corectitudinii algoritmului

Până în acest moment, nu ne-am pus problema demonstrării corectitudinii algoritmilor prezentați. Totuși, trebuie să menționăm că există o întreagă ramură a algoritmicii care se ocupă exclusiv de demonstrarea corectitudinii algoritmilor. În general, demonstrarea corectitudinii unui algoritm este un proces

destul de laborios, și, în majoritatea aplicațiilor practice concrete, este mai adecvată testarea comportamentului pe mulțimi de date reprezentative decât demonstrarea riguroasă a corectitudinii. Există totuși situații în care trebuie să ne asigurăm că o anumită secvență critică din cadrul unui program face *exact* ceea ce intenționăm, indiferent de setul de date care este primit la intrare. De exemplu, pentru un sistem de operare, este esențial ca algoritmul de planificare al proceselor să funcționeze corect, indiferent de numărul și natura proceselor care trebuie planificate. Oricâte seturi de date am alege pentru a testa algoritmul, rămâne posibilitatea ca să fi scăpat din vedere o anumită configurație pentru care algoritmul ar funcționa greșit. Astfel, singura posibilitate de a ne convinge că algoritmul va funcționa corect indiferent de setul de date de la intrare este să *demonstrăm* riguros corectitudinea lui.

Pentru algoritmul de față am ales să îi demonstrăm corectitudinea, deoarece pe de o parte ea este ilustrativă pentru o întreagă clasă de probleme, iar pe de altă parte demonstrația nu este dificilă.

**Teorema 13.1.1** *Algoritmul SELECT-SPECTACOLE-GREEDY furnizează soluția optimă (număr maxim de spectacole) pentru problema selectării activităților.*

*Demonstrație:* Fie  $S = \{1, 2, \dots, n\}$  mulțimea activităților care trebuie planificate, ordonate crescător după timpul de terminare. În consecință, activitatea 1 se termină cel mai devreme. Vom arăta că există o soluție optimă care începe cu activitatea 1.

Să presupunem că avem o soluție  $A \subseteq S$  optimă pentru o instanță a problemei. Pentru simplitate, presupunem că activitățile din  $A$  sunt ordonate după timpul de terminare. Dacă primul spectacol din  $A$  este chiar 1, atunci demonstrația este încheiată. Dacă primul spectacol din  $A$  nu este 1, atunci înlocuim primul spectacol cu spectacolul 1, obținând evident o soluție corectă, deoarece spectacolul 1 se va termina mai devreme decât primul spectacol din  $A$ . Am arătat astfel că există o soluție optimă pentru  $S$  care începe cu activitatea 1.

Mai mult, odată ce este făcută alegerea activității 1, problema se reduce la determinarea soluției optime pentru activitățile din  $S$  care sunt compatibile cu activitatea 1. Fie  $S' = \{i \in S \mid s_i \geq t_1\}$  mulțimea activităților care încep după ce 1 se termină. Rezultă că dacă  $A$  este o soluție optimă pentru  $S$ , atunci  $A' = A - \{1\}$  este o soluție optimă pentru  $S'$ . Dacă nu ar fi așa, atunci ar exista o soluție optimă  $B'$  pentru  $S'$  care să aibă mai multe activități decât  $A'$ . Adăugând activitatea 1 la  $B'$ , vom obține o soluție pentru  $S$  cu mai multe activități decât soluția  $A$ , ceea ce este absurd.

Astfel, prin inducție după numărul de alegeri făcute se poate arăta că alegând primul spectacol compatibil la fiecare pas, se obține o soluție optimă.

### 13.1.2 Soluția problemei spectacolelor

Soluția problemei spectacolelor este dată în **Listing 13.1**. Metoda `selectSpectacole()` de la liniile 12-33 este transpunerea în Java a funcției *SELECT-SPECTACOLE-GREEDY*. `selectSpectacole()` primește ca parametri două șiruri reprezentând timpul de început respectiv timpul de sfârșit al fiecărui spectacol și întoarce un vector cu spectacolele care au fost planificate, în ordinea crescătoare a timpului de începere. Ordonarea crescătoare a spectacolelor este realizată de metoda `ordonare()` de la liniile 35-70, care, pentru simplitate, folosește metoda bulelor. De remarcat faptul că metoda `ordonare()` trebuie să interschimbe și valorile din șirul `s`, pentru a menține consistența cu șirul `t`.

Listing 13.1: Soluția problemei spectacolelor

```

1 import java.util.*;
2 import java.io.*;
3 import io.Reader;
4
5 /**
6  * Problema spectacolelor (selectarea activitatilor prin
7  * metoda Greedy)
8  */
9 public class Spectacole
10 {
11     /** Selectarea spectacolelor. */
12     public static Vector selectSpectacole(int[] s, int[] t)
13     {
14         Vector sol = new Vector();
15
16         if (s.length == 0) return sol;
17
18         //primul spectacol face parte din solutie
19         sol.addElement(new Integer(0));
20
21         int j = 0;
22
23         for (int i = 1; i < s.length; i++)
24         {
25             if (s[i] >= t[j])
26             {
27                 sol.addElement(new Integer(i));
28                 j = i;
29             }
30         }
31
32         return sol;
33     }
34
35     /** Ordonarea timpilor de terminare a spectacolelor. */

```

```

36 public static void ordonare(int[] s, int[] t)
37 {
38     //ordonare prin "bubble sort"
39
40     //daca interschimbam valorile din sir
41     //atunci k este 1, altfel k este 0
42     int k;
43     int aux;
44
45     do
46     {
47         //la inceput nu sunt schimbari in siruri
48         k = 0;
49
50         for (int i = 0; i < t.length - 1; i++)
51         {
52             if (t[i] > t[i + 1])
53             {
54                 //interschimbam valorile din s
55                 aux = s[i];
56                 s[i] = s[i + 1];
57                 s[i + 1] = aux;
58
59                 //interschimbam valorile din t
60                 aux = t[i];
61                 t[i] = t[i + 1];
62                 t[i + 1] = aux;
63
64                 //au fost facute schimbari in siruri
65                 k = 1;
66             }
67         }
68     }
69     while (k == 1);
70 }
71
72 /** Programul principal. */
73 public static void main(String[] args)
74 {
75     //citirea numarului de spectacole
76     System.out.print("Introduceti numarul de spectacole: ");
77     int n = Reader.readInt();
78
79     //citirea timpilor de incepere si terminare ai spectacolelor
80     System.out.println("Introduceti timpii de incepere si " +
81         "terminare ai spectacolelor:");
82     int[] s = new int[n];
83     int[] t = new int[n];
84
85     for(int i = 0; i < s.length; i++)

```



```

86     {
87         System.out.print("s[" + i + "] = ");
88         s[i] = Reader.readInt();
89         System.out.print("t[" + i + "] = ");
90         t[i] = Reader.readInt();
91     }
92
93     //ordonarea crescatoare a spectacolelor in functie de
94     //timpul de terminare a spectacolelor
95     ordonare(s, t);
96
97     //se selecteaza prin metoda greedy spectacolele
98     Vector sol = selectSpectacole(s, t);
99
100    //afisarea rezultatelor
101    System.out.println("Organizarea spectacolelor:");
102    for (int i = 0; i < sol.size(); i++)
103    {
104        int id = ((Integer) sol.elementAt(i)).intValue();
105
106        System.out.println(s[id] + " " + t[id]);
107    }
108 }
109 }
110 }

```

## 13.2 Elemente ale strategiei Greedy

Un algoritm Greedy determină o soluție optimă a unei probleme în urma unei succesiuni de alegeri. La fiecare moment de decizie din algoritm este aleasă opțiunea care pare a fi cea mai potrivită. Această *strategie euristică* nu produce întotdeauna soluția optimă, dar există și cazuri când aceasta este obținută, cum ar fi în cazul problemei selectării activităților. În acest paragraf vom prezenta câteva proprietăți generale ale metodei Greedy.

Cum se poate decide dacă un algoritm Greedy poate rezolva o problemă particulară de optimizare? În general nu există o modalitate de a stabili acest lucru, dar există anumite caracteristici pe care le au majoritatea problemelor care se rezolvă prin tehnici *Greedy*: *proprietatea de alegere Greedy* și *substructura optimă*.

În cazul general o problemă de tip Greedy, are următoarele componente:

- o *mulțime de candidați* (lucrări de planificat, vârfuri ale grafului etc);
- o funcție care verifică dacă o anumită mulțime de candidați constituie o *soluție posibilă* (nu neapărat optimă) a problemei;

- o funcție care verifică dacă o mulțime de candidați este *fezabilă*, adică dacă este posibil să completăm această mulțime astfel încât să obținem o soluție posibilă (nu neapărat optimă) a problemei (verifică dacă planificarea este formată din activități care nu se suprapun, etc.);
- o *funcție de selecție* care indică la orice moment care este cel mai promițător dintre candidații încă nefolosiți (se alege spectacolul compatibil care se termină cel mai repede);
- o *funcție obiectiv* care dă valoarea unei soluții (numărul de lucrări planificate, timpul necesar executării tuturor lucrărilor într-o anumită ordine, lungimea drumului pe care l-am găsit, etc.); aceasta este funcția pe care urmărim să o optimizăm (minimizăm/maximizăm).

Pentru a rezolva o problemă de optimizare cu Greedy, căutăm o soluție posibilă care să optimizeze valoarea funcției obiectiv. Un algoritm Greedy construiește soluția pas cu pas. Inițial, mulțimea candidaților selectați este vidă. La fiecare pas, încercăm să adăugăm acestei mulțimi cel mai promițător candidat, conform funcției de selecție. Dacă, după o astfel de adăugare, mulțimea de candidați selectați nu mai este fezabilă, eliminăm ultimul candidat adăugat, iar acesta nu va mai fi niciodată considerat. Dacă, după adăugare, mulțimea de candidați selectați este fezabilă, ultimul candidat adăugat va rămâne de acum încolo în ea. De fiecare dată când lărgim mulțimea candidaților selectați, verificăm dacă această mulțime nu constituie o soluție posibilă a problemei noastre. Dacă algoritmul Greedy funcționează corect, prima soluție găsită va fi totodată o soluție optimă a problemei. Soluția optimă nu este în mod necesar unică: se poate ca funcția obiectiv să aibă aceeași valoare optimă pentru mai multe soluții posibile. Descrierea în pseudocod a unui algoritm Greedy general este:

```

funcție greedy( $C$ ) //  $C$  este mulțimea candidaților
 $S \leftarrow \emptyset$  //  $S$  este mulțimea în care construim soluția
cât timp not soluție( $S$ ) și  $C \neq \emptyset$ 
     $x \leftarrow$  un element din  $C$  care maximizează select( $x$ )
     $C \leftarrow C - \{x\}$ 
    dacă fezabil( $S \cup \{x\}$ ) atunci  $S \leftarrow S \cup \{x\}$ 
dacă soluție( $S$ ) atunci
    return  $S$ 
altfel
    return "nu există soluție"

```

Este de înțeles acum de ce un astfel de algoritm se numește "lacom" (am putea să-l numim și "nechibzuit"). La fiecare pas, *greedy()* alege cel mai bun candidat la momentul respectiv, fără să-i pese de viitor și fără să se răzgândească. Dacă un candidat este inclus în soluție, el rămâne acolo; dacă un candidat este exclus din soluție, el nu va mai fi niciodată reconsiderat. Asemenea unui întreprinzător care urmărește câștigul imediat în dauna celui de perspectivă, un algoritm Greedy acționează simplist. Totuși, ca și în afaceri, o astfel de metodă poate da rezultate foarte bune tocmai datorită simplității ei. Funcția de selectare este în general derivată, ca și funcția de continuare de la metoda backtracking, din funcția obiectiv. Să identificăm acum elementele strategiei Greedy pentru următoarea problemă:

*Să se scrie un algoritm care este capabil să dea o anumită sumă, reprezentând restul unui client, folosind un număr cât mai mic de monezi, având valorile de 1, 5 și 25 de unități.*

Elementele strategiei Greedy sunt:

- *Candidații*: mulțimea inițială de monezi de 1, 5 și 25 unități, în care presupunem că din fiecare tip de monedă avem o cantitate nelimitată;
- *O soluție posibilă*: valoarea totală a unei astfel de mulțimi de monezi selectate trebuie să fie exact valoarea pe care trebuie să o dăm ca rest;
- *O mulțime fezabilă*: valoarea totală a unei astfel de mulțimi de monezi selectate nu este mai mare decât valoarea pe care trebuie să o dăm ca rest;
- *Funcția de selecție*: se alege cea mai mare monedă din mulțimea de candidați rămasă;
- *Funcția obiectiv*: numărul de monezi folosite în soluție; se dorește minimizarea acestui număr.

Se poate demonstra că algoritmul Greedy va găsi în acest caz mereu soluția optimă - restul cu un număr minim de monezi. Pe de altă parte, presupunând că există și monezi de 12 unități sau că unele din tipurile de monezi lipsesc din mulțimea inițială de candidați, se pot găsi contraexemple pentru care algoritmul nu găsește soluția optimă, sau nu găsește nici o soluție cu toate că există una.

Evident, soluția optimă se poate găsi și încercând toate combinațiile posibile de monezi (folosind *backtracking*), dar soluția ar avea în acest caz o complexitate exponențială, în timp ce complexitatea algoritmului Greedy este liniară.

Un algoritm Greedy nu duce deci întotdeauna la soluția optimă sau la o soluție. Este doar un principiu general, urmând ca pentru fiecare caz în parte să determinăm dacă obținem sau nu soluția optimă.

Să vedem acum care sunt cele două caracteristici esențiale pe care trebuie să le respecte o problemă pentru a putea fi abordată folosind Greedy.

### 13.2.1 Proprietatea de alegere Greedy

Prima caracteristică a unei probleme de tip Greedy este *proprietatea alegerii Greedy*, adică se poate ajunge la o soluție optimă global, realizând alegeri (Greedy) *optime local*. În procesul de construcție a unei soluții din cadrul unui algoritm Greedy se realizează o alegere care pare a fi cea mai bună la momentul respectiv. Alegerea realizată de un algoritm Greedy poate depinde de alegerile făcute până în acel moment, dar nu ia în calcul niciodată alegerile care pot fi făcute ulterior.

Desigur, trebuie să demonstrăm că o alegere Greedy la fiecare pas conduce la o soluție optimă global, dar aceasta este o problemă mai delicată. De obicei, demonstrația examinează o soluție optimă global. Apoi se arată că soluția poate fi modificată astfel încât la fiecare pas este realizată o alegere Greedy, iar această alegere reduce problema la una similară dar de dimensiuni mai reduse. Se aplică apoi principiul inducției matematice pentru a arăta că o alegere Greedy poate fi utilizată la fiecare pas. Faptul că o alegere Greedy conduce la o problemă de dimensiuni mai mici reduce demonstrația corectitudinii la demonstrarea faptului că o soluție optimă trebuie să evidențieze o substructură optimă.

### 13.2.2 Substructură optimă

O problemă evidențiază o *substructură optimă* dacă o soluție optimă a problemei conține soluții optime ale subproblemelor. Această proprietate este cheia pentru aplicarea programării dinamice sau a unui algoritm Greedy. Ca exemplu al unei structuri optime, să ne reamintim demonstrația corectitudinii algoritmului pentru problema selectării spectacolelor, unde se arată că dacă o soluție optimă  $A$  a problemei selectării activităților începe cu activitatea 1, atunci mulțimea activităților  $A' = A - \{1\}$  este o soluție optimă pentru problema selectării activităților  $S' = \{i \in S \mid s_i \geq t_1\}$ .

Cu proprietatea de substructură optimă ne vom întâlni în capitolul următor, în care vom prezenta cum se pot rezolva problemele care o respectă (fără a respecta și proprietatea de alegere Greedy) folosind metoda programării dinamice.

### 13.3 Minimizarea timpului mediu de așteptare

O singură stație de servire (procesor, pompă de benzină etc) trebuie să satisfacă cererile a  $n$  clienți. Timpul de servire necesar fiecărui client este cunoscut în prealabil: pentru clientul  $i$  este necesar un timp  $t_i, i = 1, n$ . Dorim să minimizăm timpul total de așteptare:

$$T = \sum_{i=1}^n (\text{timpul de așteptare pentru clientul } i)$$

Ceea ce este același lucru cu a minimiza timpul mediu de așteptare, care este  $\frac{T}{n}$ .

De exemplu, dacă avem trei clienți cu  $t_1 = 5, t_2 = 10, t_3 = 3$ , sunt posibile șase ordini de servire:

Ordine			Timpul (T)
1	2	3	$5+(5+10)+(5+10+3)$
1	3	2	$5+(5+3)+(5+3+10)$
2	1	3	$10+(10+5)+(10+5+3)$
2	3	1	$10+(10+3)+(10+3+5)$
3	1	2	$3+(3+5)+(3+5+10)$ optim
3	2	1	$3+(3+10)+(3+10+5)$

În primul caz, clientul 1 este servit primul, clientul 2 așteaptă până este servit clientul 1 și apoi este servit, clientul 3 așteaptă până sunt serviți clienții 1, 2 și apoi este servit. Timpul total de așteptare a celor trei clienți este 38.

Timpul total minim de așteptare este obținut în al cincilea caz și are valoarea 29.

Algoritmul Greedy este foarte simplu - la fiecare pas se selectează clientul cu timpul minim de servire din mulțimea de clienți rămasă. Vom demonstra că acest algoritm este optim. Fie  $I = (i_1 i_2 \dots i_n)$  o permutare oarecare a întregilor  $\{1, 2, \dots, n\}$ . Dacă servirea are loc în ordinea  $I$ , avem:

$$T(I) = t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \dots = nt_{i_1} + (n-1)t_{i_2} + \dots = \sum_{k=1}^n (n-k+1)t_{i_k}$$

Presupunem acum că  $I$  este astfel ales încât putem găsi doi întregi  $a < b$  cu

$$t_{i_a} > t_{i_b}$$

deci există un client ( $b$ ) care necesită un timp mai lung de servire, și care este servit înainte (de  $a$ ).

Interschimbăm pe  $i_a$  cu  $i_b$  în  $I$ ; cu alte cuvinte, clientul care a fost servit al  $b$ -lea va fi servit acum al  $a$ -lea și invers. Obținem o nouă ordine de servire  $I'$ , care este de preferat deoarece

$$T(I') = (n - a + 1)t_{i_b} + (n - b + 1)t_{i_a} + \sum_{k=1, k \neq a, b}^n (n - k + 1)t_{i_k}$$

$$T(I) - T(I') = (n - a + 1)(t_{i_a} - t_{i_b}) + (n - b + 1)(t_{i_b} - t_{i_a}) = (b - a)(t_{i_a} - t_{i_b}) > 0$$

Aplicând succesiv pasul de mai sus se obține o permutare optimă, de tipul  $J = (j_1, j_2, \dots, j_n)$  pentru care avem:

$$t_{j_1} \leq t_{j_2} \leq \dots \leq t_{j_n}.$$

Prin metoda Greedy, selectând permanent clientul cu timpul cel mai mic de deservire, obținem deci întotdeauna planificarea optimă a clienților. Problema poate fi generalizată și pentru un sistem cu mai multe stații de servire.

Implementarea algoritmului se reduce la o banală ordonare a clienților crescător după timpul de deservire și este prezentată în **Listing 13.2**.

Listing 13.2: Soluția problemei minimizării timpului de așteptare

```

1 import java.io.*;
2 import io.Reader;
3
4 /**
5  * Program pentru minimizarea timpului de asteptare al unui
6  * client pentru a fi deservit de o statie (metoda Greedy).
7  */
8 public class MinimTimp
9 {
10     /** Ordonarea timpilor de asteptare ai clientilor. */
11     public static void ordonare(int[] t)
12     {
13         //ordonare prin "bubble sort"
14
15         //daca interschimbam valorile din sir
16         //atunci k este 1, altfel k este 0
17         int k;
18         int aux;
19
20         do
21         {
22             //la inceput nu sunt schimbari in siruri
23             k = 0;
24
25             for (int i = 0; i < t.length - 1; i++)
26             {
27                 if (t[i] > t[i + 1])
28                 {
29                     //interschimbam valorile din t
30                     aux = t[i];

```

```

31         t[i] = t[i + 1];
32         t[i + 1] = aux;
33
34         //au fost facute schimbări în sir
35         k = 1;
36     }
37 }
38 }
39 while (k == 1);
40 }
41
42 /** Calculează timpul total minim de așteptare.*/
43 public static int calculTimpMinim(int[] t)
44 {
45     int s = 0;
46
47     System.out.println("Ordinea de deservire este:");
48
49     for (int i = 1; i <= t.length; i++)
50     {
51         s += (t.length - i + 1) * t[i - 1];
52
53         System.out.print(t[i - 1] + " ");
54     }
55     System.out.println();
56
57     return s;
58 }
59
60
61 /** Programul principal.*/
62 public static void main(String[] args)
63 {
64     //citirea timpului de așteptare al fiecărui client
65     System.out.println("Timpii de așteptare pentru " +
66         "clienți (pe aceeași linie):");
67     int[] t = Reader.readIntArray();
68
69     //ordonarea timpilor de așteptare ai clienților
70     ordonare(t);
71
72     System.out.println(
73         "Timpul total de așteptare are valoarea minimă = " +
74         calculTimpMinim(t));
75
76 }
77 }

```

## 13.4 Interclasarea optimă a mai multor șiruri ordonate

Să presupunem că avem două șiruri  $S_1$  și  $S_2$ , de lungime  $m$  și  $n$ , ordonate crescător și că dorim să obținem prin interclasarea lor un șir ordonat crescător,  $S$ , care conține exact elementele din cele două șiruri. Dacă interclasarea are loc prin plasarea elementelor din  $S_1$  și  $S_2$  în noul șir,  $S$ , atunci numărul deplasărilor este  $m + n$ .

Generalizând, să considerăm acum  $n$  șiruri  $S_1, S_2, \dots, S_n$ , fiecare șir  $S_i, i = 1, n$ , fiind format din  $q_i$  elemente ordonate crescător (vom numi  $q_i$  *lungimea* lui  $S_i$ ). Ne propunem să obținem șirul  $S$  ordonat crescător, conținând exact elementele din cele  $n$  șiruri. Vom realiza acest lucru prin interclasări succesive de câte două șiruri. Problema interclasării optime a mai multor șiruri ordonate constă în determinarea *ordinii* optime în care trebuie efectuate aceste interclasări, astfel încât numărul total de operații (deplasări) să fie cât mai mic. Exemplul de mai jos ne arată că problema astfel formulată nu este banală, adică nu este indiferent în ce ordine se fac interclasările.

**Exemplu:** Fie șirurile  $S_1, S_2, S_3$  de lungimi  $q_1 = 30, q_2 = 20, q_3 = 10$ . Dacă interclasăm pe  $S_1$  cu  $S_2$ , iar rezultatul îl interclasăm cu  $S_3$ , numărul total al deplasărilor este  $(30 + 20) + (50 + 10) = 110$ . Dacă interclasăm pe  $S_3$  cu  $S_2$ , iar rezultatul îl interclasăm cu  $S_1$ , numărul total al deplasărilor este  $(10 + 20) + (30 + 30) = 90$ , deci cu 20 de operații mai puțin.

Atașăm fiecărei strategii de interclasare câte un arbore binar în care valoarea fiecărui vârf este dată de lungimea șirului pe care îl reprezintă. Dacă șirurile  $S_1, S_2, \dots, S_6$  au lungimile  $q_1 = 30, q_2 = 10, q_3 = 20, q_4 = 30, q_5 = 50, q_6 = 10$ , două astfel de strategii de interclasare sunt reprezentate prin arborii din **Figura 13.1**.

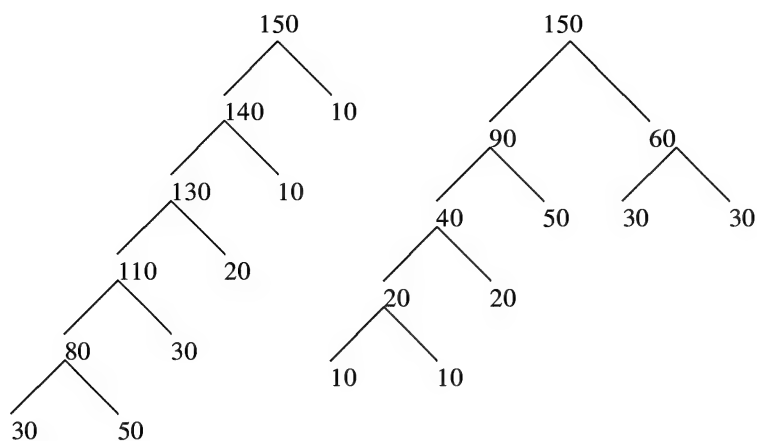
Observăm că fiecare arbore are 6 vârfuri terminale, corespunzând celor 6 șiruri inițiale și 5 vârfuri neterminale, corespunzând celor 5 interclasări care definesc strategia respectivă. Numerotăm vârfurile în felul următor: vârful terminal  $i$  ( $i \in 1, 2, 3, 4, 5, 6$ ), va corespunde șirului  $S_i$ , iar vârfurile neterminale se numerotează de la 7 la 11 în ordinea obținerii interclasărilor respective (**Figura 13.2**).

Strategia Greedy apare în **Figura 13.1** (arborile din partea dreaptă) și constă în a interclasa mereu cele mai scurte două șiruri disponibile la momentul respectiv.

Pentru a interclasa șirurile  $S_1, S_2, \dots, S_n$ , de lungimi  $q_1, q_2, \dots, q_n$ , obținem pentru fiecare strategie câte un arbore binar cu  $n$  vârfuri terminale numerotate



Figura 13.1: Reprezentarea strategiilor de interclasare.



de la 1 la  $n$  și  $n - 1$  vârfuri neterminale numerotate de la  $n + 1$  la  $2n - 1$ . Definim pentru un arbore oarecare  $A$  de acest tip lungimea externă ponderată:

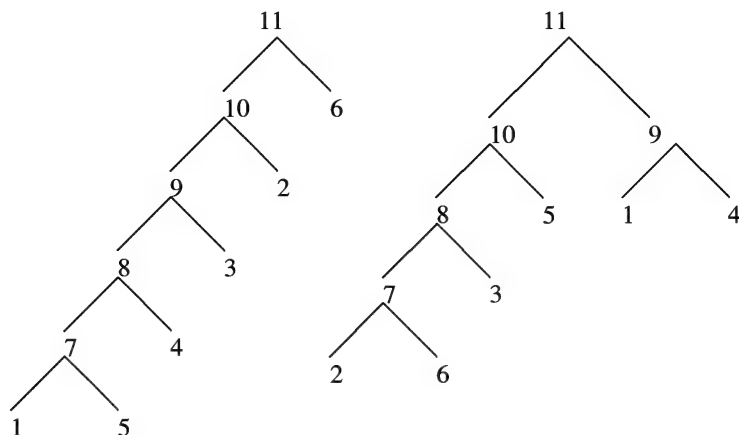
$$L(A) = \sum_{i=1}^n a_i q_i$$

unde  $a_i$  este adâncimea vârfului  $i$ . Este ușor de observat că numărul total de deplasări de elemente pentru strategia corespunzătoare lui  $A$  este chiar  $L(A)$ . Soluția optimă a problemei noastre este atunci arborele (strategia) pentru care lungimea externă ponderată este minimă.

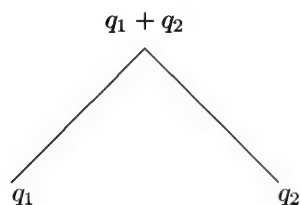
**Teorema 13.4.1** Prin metoda Greedy, în care se interclasează la fiecare pas cele două șiruri de lungime minimă, se obține șirul  $s$  cu un număr minim de operații.

*Demonstrație:* Demonstrăm prin inducție. Pentru  $n = 1$ , proprietatea este verificată. Presupunem că proprietatea este adevărată pentru  $n - 1$  șiruri. Fie  $A$

Figura 13.2: Numerotarea vârfurilor arborilor din Figura 13.1



arborele strategiei Greedy de interclasare a  $n$  șiruri de lungime  $q_1 \leq q_2 \leq \dots \leq q_n$ . Fie  $B$  un arbore cu lungimea externă ponderată minimă, corespunzător unei strategii optime de interclasare a celor  $n$  șiruri. În arborele  $A$  apare subarborele:



reprezentând prima interclasare făcută conform strategiei Greedy. În arborele  $B$ , fie un vârf neterminal de adâncime maximă. Cei doi fii ai acestui vârf sunt atunci două vârfuri terminale  $q_j$  și  $q_k$ . Fie  $B'$  arborele obținut din  $B$  schimbând între ele vârfurile  $q_1$  și  $q_j$ , respectiv  $q_2$  și  $q_k$ . Evident,  $L(B') \leq L(B)$ . Deoarece  $B$  are lungimea externă ponderată minimă, rezultă că  $L(B) = L(B')$ . Eliminând din  $B'$  vârfurile  $q_1$  și  $q_2$ , obținem un arbore  $B''$  cu  $n - 1$  vârfuri

terminale  $q_1 + q_2, q_3, \dots, q_n$ . Arborele  $B'$  are lungimea externă ponderată minimă și  $L(B') = L(B) + (q_1 + q_2)$ . Rezultă că și  $B$  are lungimea externă ponderată minimă. Atunci, conform ipotezei inducției, avem  $L(B) = L(A')$  unde  $A'$  este arborele strategiei Greedy de interclasare a șirurilor de lungime  $q_1 + q_2, q_3, \dots, q_n$ . Cum  $A$  se obține din  $A'$  atașând la vârful  $q_1 + q_2$  fiii  $q_1$  și  $q_2$ , iar  $B'$  se obține în același mod din  $B''$ , rezultă că  $L(A) = L(B') = L(B)$ . Proprietatea este deci adevărată pentru orice  $n$ .

Deoarece algoritmul alege la fiecare pas șirurile disponibile de lungime minimă, structura de date adecvată pentru a reține șirurile este coada de prioritate, prezentată în paragraful 10.7. Implementarea algoritmului este simplă și o lăsăm ca exercițiu.

## Rezumat

În acest capitol am prezentat metoda Greedy, care se poate aplica problemelor care respectă substructura optimă și principiul alegerii Greedy. În cazul acestei metode, soluția se construiește succesiv, la fiecare pas realizând o alegere optimă local, fără a reveni asupra deciziilor anterioare. Soluția construită astfel, va fi o soluție optimă a problemei de rezolvat. Am văzut modul în care se poate demonstra corectitudinea unui algoritm utilizând un procedeu general, care se poate aplica la mulți algoritmi de acest tip.

## Noțiuni fundamentale

**funcție de selecție:** funcție care indică cel mai promițător dintre candidații încă nefolosiți la un moment dat.

**proprietatea de alegere Greedy:** se poate ajunge la o soluție optimă global, realizând alegeri (Greedy) *optime locale*.

**problemă de optimizare:** problemă în care se cere minimizarea sau maximizarea unei funcții obiectiv.

**substructură optimă** (principiul optimalității): o problemă evidențiază o *substructură optimă* dacă o soluție optimă a problemei conține soluții optime ale subproblemelor.

## Erori frecvente

1. Nu încercați să aplicați această metodă orbește, fără a verifica dacă problema respectă substructura optimă și proprietatea de alegere Greedy.

2. Faptul că ați găsit un algoritm Greedy care funcționează corect pe unele exemple, nu înseamnă că metoda va da soluția optimă pentru orice date de intrare. Pentru a fi siguri de aceasta, demonstrați corectitudinea algoritmului.

## Exerciții

### Teorie

1. Calculați complexitatea rezolvării problemei spectacolelor din paragraful 13.1, **Listing 13.1**.
2. Găsiți o modalitate de a reduce complexitatea rezolvării problemei anterioare la  $O(n \log n)$ .
3. În rezolvarea problemei interclasării optime din paragraful 13.4 am recomandat utilizarea unei cozi de priorități. Care este complexitatea algoritmului în acest caz? Dar dacă în loc de o coadă de priorități folosim un arbore binar de căutare?

### În practică

1. Rezolvarea problemei spectacolelor din paragraful 13.1 folosește metoda bulelor pentru a ordona spectacolele în ordinea crescătoare a timpului de terminare. Înlocuiți metoda bulelor cu o ordonare mai eficientă.
2. Rezolvați problema interclasării optime din paragraful 13.4 folosind
  - (a) O listă înlănțuită;
  - (b) O coadă de priorități;
  - (c) Un arbore binar de căutare.
3. (*Problema rucsacului*) Avem la dispoziție un rucsac de capacitate  $M$  și  $n$  obiecte diferite (câte unul din fiecare) cu costurile  $c_i$  și greutatea  $g_i$ . Scrieți un algoritm care așează aceste obiecte în rucsac astfel încât costul total să fie maxim. Suma greutateilor obiectelor din rucsac nu poate depăși capacitatea rucsacului. Dacă un obiect nu încapă în rucsac, se poate lua doar o parte (fracțiune) din el.

*Indicație: Este ușor de intuit că pentru obținerea unui profit (cost) total maxim trebuie alese obiecte de greutate mică și cost mare. Demonstrația riguroasă a afirmației anterioare v-o propunem spre rezolvare.*

4. (*Problema discretă a rucsacului*) Același enunț ca la problema precedentă, cu diferența că dintr-un obiect nu se poate pune o fracțiune (un obiect fie se pune întreg, fie nu se pune). Arătați că algoritmul Greedy de la problema precedentă nu furnizează întotdeauna soluție optimă în acest caz. Găsiți un algoritm care furnizează întotdeauna soluție optimă! Ce complexitate are acest algoritm?

*Indicație: Soluția optimă pentru această problemă poate fi determinată doar prin backtracking sau programare dinamică (prezentată în capitolul următor). De exemplu, pentru  $c=(5,3,3)$ ,  $g=(3,2,2)$  și  $M=4$ , prin aplicarea algoritmului de la problema precedentă am selecta doar obiectul 1 (obiectul 2 nu ar încapa întreg în rucsac deci nu ar putea fi selectat) și am obține costul 5, în timp ce dacă am selecta obiectele 2 și 3 am obține costul 6.*

5. Găsiți o soluție Greedy pentru problema comis-voiajorului propusă la capitolul 11. Este această soluție optimă?

*Indicație: Conform strategiei greedy, vom construi ciclul pas cu pas, adăugând la fiecare iterație cea mai scurtă muchie disponibilă cu următoarele proprietăți:*

- nu formează un ciclu cu muchiile deja selectate (exceptând pentru ultima muchie aleasă, care completează ciclul);
- nu există încă două muchii deja selectate, astfel încât cele trei muchii să fie incidente în același vârf.

*Acest algoritm nu numai că nu furnizează soluția optimă, dar în multe situații nici măcar nu găsește o soluție.*

## 14. Programare dinamică

Mulți oameni preferă să tolereze o problemă pe care nu o pot rezolva, decât o soluție pe care nu o pot înțelege.

---

Woolsey and Swanson

În capitolul anterior am văzut cum putem aplica metoda Greedy problemelor de optimizare care respectă principiul optimalității (substructură optimă) și principiul alegerii Greedy. Programarea dinamică este o altă metodă de elaborare a algoritmilor care se aplică problemelor de optimizare care respectă principiul optimalității (fără a respecta principiul alegerii Greedy). În acest capitol vom introduce gradat noțiunile și tehnicile specifice acestei metode de elaborare a algoritmilor, considerată de mulți ca având un grad de dificultate mai mare decât celelalte metode<sup>1</sup>. Dar nu vă îngrijorați! Materialul din acest capitol poate fi cu ușurință parcurs de către oricine, datorită structurării gradate a noțiunilor prezentate.

Pe parcursul acestui capitol vom vedea:

- Cum se tratează *dinamic* problemele de recurență matematică, în vederea unei rezolvări eficiente;
- Ce este principiul optimalității;
- Cum se aplică principiul optimalității pentru a descompune problema de programare dinamică în subprobleme;
- Cum se creează relațiile de recurență care rezolvă problema de programare dinamică;

---

<sup>1</sup>De altfel, majoritatea problemelor spinoase alese pentru concursurile naționale și internaționale de informatică se rezolvă prin această metodă.

- Cum se poate reface soluția problemei pe baza calculelor efectuate pentru obținerea valorii optime;
- Câteva probleme clasice de programare dinamică.

## 14.1 Istoric și descriere

Programarea dinamică provine din domeniul cercetărilor operaționale, unde constituie un domeniu de sine stătător. Cuvântul *programare* din numele acestei metode nu are nimic de a face cu scrierea de programe pentru calculator. Matematicienii foloseau acest cuvânt pentru a descrie un set de reguli pe care oricine le poate urmări pentru a rezolva un anumit tip de problemă. Programarea dinamică a fost introdusă în anii 1950 de către matematicianul Richard Bellman, care a descris modalitatea prin care se rezolvă problemele în care trebuie luată o decizie optimă la fiecare pas. În cei mai bine de 50 de ani de la apariția acestei metode utilizarea și aplicarea programării dinamice au crescut enorm.

Programarea dinamică descompune de obicei problema de optimizare originală în subprobleme și alege cele mai bune soluții pentru subprobleme începând chiar cu cele de dimensiune minimă. Soluția optimă a problemelor de dimensiune mai mare este obținută pe baza soluțiilor optime a problemelor de dimensiune mai mică cu ajutorul unei formule de recurență care face legătura între soluții. Până în acest punct descrierea făcută nu diferă cu nimic de cea a tehnicii Divide et Impera. Ceea ce face ca programarea dinamică să fie mai specială este faptul că formula de recurență este folosită pentru a elimina toate soluțiile subproblemelor care nu pot genera soluția optimă. Mai mult, în cazul programării dinamice se rețin soluțiile subproblemelor care pot fi utilizate în calculul soluției optime.

Similarități există și cu metoda Greedy. Așa cum vom vedea, și problemele de programare dinamică trebuie să respecte *substructura optimă*, numită în această situație *principiul optimalității*. Diferența constă în faptul că în cazul programării dinamice nu se face o alegere Greedy, mergând pe optimul local, ci se iau în considerare *toate* soluțiile optime ale subproblemelor cu un ordin de mărime mai mic. Există multe probleme care admit o rezolvare atât prin metoda Greedy, cât și prin programare dinamică (problema rucsacului, problema comis-voiajorului). În aceste situații, soluția Greedy va avea o complexitate în timp mai mică, dar soluțiile găsite nu vor fi întotdeauna cele optime, în timp ce soluțiile date prin programare dinamică vor avea o complexitate în timp ceva mai mare, dar vor furniza soluția optimă.

Dificultățile care apar în utilizarea metodei programării dinamice au făcut ca această metodă să fie considerată de mulți ca fiind prea complexă și în con-

secință să evite aprofundarea și utilizarea ei. Scopul acestui capitol este tocmai acela de a forma treptat capacitatea de a analiza și rezolva cu multă ușurință orice gen de problemă de programare dinamică.

## 14.2 Primii pași în programarea dinamică

Acest paragraf este consacrat tratării unor probleme de recurență matematică. Nu putem afirma că problemele de recurență matematică se încadrează strict în categoria problemelor de programare dinamică, deoarece ele nu sunt probleme de optimizare. Totuși, modul în care vom rezolva aici recurențele introduce deja un concept esențial pentru rezolvarea problemelor de programare dinamică: tabelul<sup>2</sup>. Tot în acest paragraf vom analiza și care sunt avantajele utilizării tabelelor pentru a reține soluțiile subproblemelor în dauna soluțiilor recursive.

### 14.2.1 Probleme de recurență matematică tratate dinamic

#### Șirul lui Fibonacci

Un exemplu clasic, adesea întâlnit în lucrările care tratează recursivitatea, prezentat și în paragraful 12.1.1 (pagina 138) este calculul șirului lui Fibonacci, definit astfel:

$$F_n = F_{n-1} + F_{n-2}; F_0 = 0; F_1 = 1$$

Metoda recursivă care calculează  $F_n$  este descrisă în **Listing 14.1**. Așa cum am văzut deja în paragraful 9.4.3 (pagina 26), funcția de mai sus recalculează din nou și din nou termenii șirului, rezultând astfel un timp de lucru exponențial ( $O(\phi^n)$ ). **Figura 14.1** pune în evidență modul în care se recalculează din nou și din nou aceleași valori din cauza apelurilor recursive suprapuse.

Listing 14.1: Metoda recursivă de calcul a șirului lui Fibonacci

```
1 public static final int fibonacciRec(int n)
2 {
3     if (n == 0 || n == 1)
4     {
5         return n;
6     }
7     else
8     {
```

<sup>2</sup>Este interesant de remarcat faptul că recurgerea la tabele a dat și numele metodei. Astfel, în cercetările operaționale (domeniul din care provine această metodă) termenul “programare” se referă la un set de reguli care se aplică pe un tabel și nu la scrierea unui program pentru calculator.





---

```

1 import java.io.*;
2 import io.Reader;
3
4 public class Fibonacci
5 {
6     public static int fibonacciIter(int n)
7     {
8         // ...
9     }
10
11    public static void main(String args[])
12    {
13        System.out.print("n = ");
14        int n = Reader.readInt();
15
16        if (n < 1) return;
17
18        System.out.println("Termenul " + n + " din sirul " +
19            " Fibonacci este: " + fibonacciIter(n - 1));
20    }
21 }

```

Clasa `Fibonacci` din **Listing 14.3** citește un număr întreg  $n$  de la tastatură și calculează valoarea șirului lui Fibonacci la poziția  $n$  (altfel spus, termenul de ordin  $n$  al șirului) folosind metoda `fibonacciIter()`.

### Calculul combinărilor

În cadrul calculului șirului lui Fibonacci din paragraful anterior am utilizat un tabel unidimensional pentru a reține valorile șirului. În exemplul care urmează vom prezenta o problemă în rezolvarea căreia este necesară utilizarea unui tablou bidimensional (utilizarea tablourilor bidimensionale este mai frecventă în rezolvarea problemelor de programare dinamică).

Este cunoscut faptul că prin combinări de  $n$  luate câte  $k$  (notate  $C(n, k)$ ) se înțelege numărul de submulțimi care conțin  $k$  elemente ale unei mulțimi cu  $n$  elemente. Formula matematică pentru combinări de  $n$  luate câte  $k$  este:

$$C(n, k) = \frac{n!}{k!(n-k)!}$$

Nu este însă avantajos ca numerele  $C(n, k)$  să fie calculate direct, folosind formula de mai sus. Formula de recurență  $C(n, k) = C(n-1, k) + C(n-1, k-1)$  este mult mai rapidă, deoarece utilizează numai adunări și elimină operațiile de împărțire și înmulțire care sunt mai costisitoare în timp. Metoda

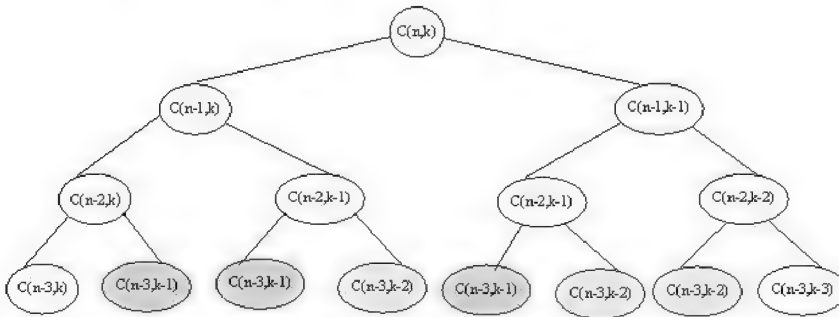
Listing 14.4: Calculul recursiv al combinărilor

```

1 public static int combRec(int n, int k)
2 {
3     if (n == k || k == 0)
4     {
5         return 1;
6     }
7     else
8     {
9         return combRec(n - 1, k) + combRec(n - 1, k - 1);
10    }
11 }

```

Figura 14.2: Reprezentarea arborelui rezultat în urma apelurilor recursive generate de recurența pentru calculul combinărilor (se observă recalcularea inutilă a valorilor  $C(n-3, k-1)$  și  $C(n-3, k-2)$ ).



`combRec()` din **Listing 14.4** reprezintă implementarea recursivă directă a formulei de recurență anterioare.

Această metodă, deși este foarte elegantă și compactă, suferă de aceleași recalculări inutile ale anumitor valori ca și algoritmul recursiv pentru calculul șirului lui Fibonacci din **Listing 14.1**. Recalculările inutile sunt puse în evidență de **Figura 14.2**, în care am reprezentat primele 3 nivele din arborele generat prin calculul recursiv al combinărilor.

Notând cu  $t_{nk}$  numărul de operații efectuate pentru a calcula  $C(n, k)$  folosind metoda de mai sus, obținem relația de recurență:

$$t_{nk} = t_{n-1,k} + t_{n-1,k-1}; \quad t_{n0} = t_{nn} = 1.$$

Iterând recurența de mai sus se obține cu ușurință faptul că timpul de calcul

Figura 14.3: Tabelul utilizat pentru calculul iterativ al combinărilor. Se observă că acest tabel nu este altceva decât triunghiul lui Pascal.

$C(0, 0)$				
$C(1, 0)$	$C(1, 1)$			
$C(2, 0)$	$C(2, 1)$	$C(2, 2)$		
$C(3, 0)$	$C(3, 1)$	$C(3, 2)$	$C(3, 3)$	
$C(4, 0)$	$C(4, 1)$	$C(4, 2)$	$C(4, 3)$	$C(4, 4)$

Listing 14.5: Calculul iterativ al combinărilor

```

1 public static int combIter(int n, int k)
2 {
3     int[][] comb = new int[n + 1][n + 1];
4     for (int i = 0; i <= n; ++i)
5     {
6         comb[i][0] = comb[i][i] = 1;
7         for (int j = 1; j < i; ++j)
8         {
9             comb[i][j] = comb[i - 1][j] + comb[i - 1][j - 1];
10        }
11    }
12    return comb[n][k];
13 }

```

pentru  $t_{nk}$  este în  $O(2^n)$ .

Pentru a evita repetarea inutilă a calculelor, vom folosi din nou un tabel în care în celula  $(i, j)$  se va păstra valoarea lui  $C(i, j)$  (indicele  $i$  trebuie să fie mai mare sau egal cu  $j$ , deci tabelul nostru va avea elemente doar sub diagonala principală). Examinând **Figura 14.3** se constată că tabelul construit de noi nu este altceva decât triunghiul lui Pascal.

Metoda `combIter()` din **Listing 14.5** realizează calculul iterativ al combinărilor folosind un tabel (triunghiul lui Pascal). Complexitatea acestei metode este  $O(n^2)$ , deci ea este net superioară ca și eficiență metodei `combRec()` din **Listing 14.4**.

În cazul șirului lui Fibonacci am evaluat termenii începând cu cei de grad inferior și terminând cu cei de grad superior. În cazul combinărilor am calculat treptat mai întâi combinări de un element, apoi combinări de două elemente etc., deoarece  $C(n, k)$  depinde direct de  $C(n - 1, k)$  și  $C(n - 1, k - 1)$ , deci de două valori aflate pe linia anterioară în tabelul reprezentând triunghiul lui Pascal. O idee importantă care reiese de aici cu ușurință este următoarea:

*“Întotdeauna în cazul unei probleme de programare dinamică este necesar ca în momentul în care se rezolvă o problemă, toate sub-problemele ei să fi fost deja rezolvate.”*

Strategia de rezolvare pentru problemele de programare dinamică decurge astfel normal din această observație:

- se rezolvă mai întâi problemele de dimensiune mică pentru care soluția este evidentă (în cazul nostru acestea sunt  $F_0 = 0$ ,  $F_1 = 1$ , respectiv  $C(i, 0) = 1$ );
- pe baza supproblemelor de dimensiune inferioară se rezolvă subproblema cu un ordin de dimensiune mai mare (în cazul nostru, pe baza lui  $F_{i-1}$  și  $F_{i-2}$  se calculează  $F_i$ , respectiv pe baza lui  $C(i-1, j)$ ,  $C(i-1, j-1)$  se calculează  $C(i, j)$ ).

Clasa `CombinariIt` din **Listing 14.6** reprezintă un exemplu de simplu de utilizare a metodei `combIter()` pentru a calcula combinările a două numere citite de la tastatură.

### Problema parantezelor

Cele două probleme prezentate anterior sunt relativ simple, deoarece recurența care trebuie rezolvată apare explicit în enunț. Atât pentru șirul lui Fibonacci, cât și pentru calculul combinărilor, ni se furnizează explicit relația de recurență pe care trebuie să o rezolvăm. Problema parantezelor este un exemplu de problemă în care recurența care trebuie rezolvată nu apare explicit în enunț. Rămâne în sarcina noastră să construim relația de recurență care rezolvă această problemă. Din acest punct de vedere, problema parantezelor se apropie mai mult de problemele reale de programare dinamică, deoarece nici acolo recurența care trebuie rezolvată nu este dată explicit. Mai mult, am putea chiar afirma că principala dificultate în cazul problemelor de programare dinamică constă, mai ales la începători, în formularea corectă a relației de recurență care rezolvă problema. Așadar, dificultatea principală în cazul problemelor de programare dinamică este construirea recurenței, scrierea programului pentru rezolvarea propriu-zisă a recurenței fiind în cele mai multe cazuri banală.

Enunțul problemei parantezelor este simplu:

*Dându-se un număr natural  $n > 0$ , se cere să se determine numărul de șiruri având  $2 \cdot n$  paranteze care se închid corect.*

Listing 14.6: Soluția completă a problemei combinărilor

```

1 import java.io.*;
2 import io.Reader;
3
4 public class CombinariIt
5 {
6     public static int combinariIter(int n, int k)
7     {
8         // ...
9     }
10
11    public static void main(String args[])
12    {
13        System.out.print("n = ");
14        int n = Reader.readInt();
15        System.out.print("k = ");
16        int k = Reader.readInt();
17
18        if (n < 1 || k < 0 || k > n)
19        {
20            System.out.println("n trebuie sa fie >= 1");
21            System.out.println("k trebuie sa fie >= 0 si " +
22                "mai mic decat n");
23            return;
24        }
25
26        System.out.println("Comb ( " + n + " , " + k + " ) = " +
27            combinariIter(n, k));
28    }
29 }

```

De exemplu, pentru  $n = 2$ , șirurile închise corect sunt  $()()$  și  $(())$ , deci răspunsul este 2.

Soluția (celebră) a acestei probleme este reprezentată de așa-numitele numere catalane (vezi [Cormen], pag. 261). Totuși, datorită scopului acestui paragraf, vom da o altă soluție a problemei, bazată pe o relație simplă de recurență.

Este cert că orice șir de paranteze închise corect începe cu o paranteză deschisă “(”. Să considerăm acum paranteza care închide această primă paranteză. Ceea ce se află între aceste două paranteze este tot un șir de paranteze care se închid corect; la fel și pentru șirul care se află în dreapta lor. Deci, un șir  $S$  de paranteze care se închid corect se poate scrie ca  $(S1)S2$ , unde  $S1$  și  $S2$  sunt alte șiruri de paranteze care se închid corect, posibil vide.

Lungimea maximă a lui  $S1$  este  $2 \cdot n - 2$  (atinsă când  $S2$  este vid), iar cea minimă este 0 (atinsă când  $S2$  are lungimea  $2 \cdot n - 2$ ). Să notăm cu  $P(n)$

numărul de șiruri având  $2n$  paranteze care se închid corect. Se observă cu ușurință că paranteza care închide prima paranteză poate fi pe oricare dintre pozițiile  $2, 4, 6, \dots, 2n$ . Astfel, dacă paranteza este pe poziția  $2$ , atunci șirul  $S_1$  este vid, iar șirul  $S_2$  va avea lungimea  $2n - 2 = 2(n - 1)$ , deci vor exista  $P(0)P(n - 1)$  șiruri de paranteze care se închid corect. Analog, dacă paranteza este pe poziția  $4$ , șirul  $S_1$  va avea lungimea  $2$ , iar șirul  $S_2$  va avea lungimea  $2n - 4 = 2(n - 2)$ , deci vom avea  $P(1)P(n - 2)$  șiruri care se închid corect. Obținem astfel recurența:

$$P(n) = \begin{cases} 1 & \text{pentru } n = 0 \\ \sum_{k=0}^{n-1} P(k)P(n - k - 1) & \text{pentru } n \geq 1 \end{cases}$$

Implementarea recurenței de mai sus este simplă și este prezentată în **Listing 14.7**.

Listing 14.7: Soluția problemei parantezelor

```

1 import java.io.*;
2 import io.Reader;
3
4 public class Paranteze
5 {
6     public static int parantezeIter(int n)
7     {
8         int[] p = new int[n + 1];
9         p[0] = 1;
10
11         for (int i = 1; i <= n; i++)
12         {
13             for (int k = 0; k <= i - 1; k++)
14             {
15                 p[i] += p[k] * p[i - k - 1];
16             }
17         }
18
19         return p[n];
20     }
21
22     public static void main(String args[])
23     {
24         System.out.print("n = ");
25         int n = Reader.readInt();
26
27         if (n < 0) return;
28
29         System.out.println("Numarul de siruri cu " + 2 * n +
30             " paranteze care se inchid corect este: " +
31             parantezeIter(n));

```

32 }  
 33 }

### Descopunerea unui număr natural ca sume distincte

Problema descompunerii unui număr natural în sume distincte este un alt exemplu în care recurența care furnizează soluția trebuie construită de către programator. Enunțul problemei este:

*Dându-se un număr natural  $n$ , se cere să se determine numărul de moduri distincte<sup>3</sup> în care acesta se poate scrie ca sumă de numere naturale nenule.*

De exemplu, numărul  $n = 5$  se poate scrie ca:

1 + 1 + 1 + 1 + 1  
 1 + 1 + 1 + 2  
 1 + 1 + 3  
 1 + 2 + 2  
 1 + 4  
 2 + 3  
 5

Astfel, pentru  $n = 5$  am obținut 7 descompuneri (5 se consideră aici ca fiind sumă; se poate modifica problema excluzând această posibilitate, dar rezolvarea se păstrează, din numărul total de descompuneri trebuind doar să fie scăzut 1).

Pentru a obține recurența pentru această problemă, vom rezolva mai întâi un caz particular:

*Dându-se un număr natural  $n$ , să se determine numărul de moduri distincte în care acesta poate fi descompus în sumă de  $m$  numere naturale.*

Să notăm cu  $S(n, m)$  numărul de moduri distincte în care  $n$  se poate scrie ca sumă de  $m$  termeni. Evident, soluția problemei originale va fi

$$\sum_{i=1}^n S(n, i)$$

Problema originală se reduce așadar la a calcula  $S(n, i)$   $i = 1, 2, \dots, n$ .

Se observă cu ușurință că  $S(i, 1) = S(i, i) = 1$  (acestea sunt condițiile inițiale). De asemenea,  $S(i, i + k) = 0$  pentru oricare  $k$  număr natural pozitiv.

<sup>3</sup>Adunarea fiind comutativă, vom considera că descompunerile  $2 + 3$  și  $3 + 2$  sunt echivalente.



Listing 14.8: Metoda de calcul a numărului de descompuneri

```

1 public static int descompuneNumar( int n )
2 {
3     int s = new int[n][n] ; //alocam memorie pentru tabelul s
4     //construim tabelul s
5     for( int i=0; i<n; ++i )
6     {
7         s[i][0] = s[i][i] = 1 ; //conditiile initiale
8         for( int j=1; j<i; ++i )
9         {
10             for( int k=0; k<j; ++k )
11             {
12                 s[i][j] += s[i-j][k] ;
13             }
14         }
15     }
16     int nrDescompuneri = 0 ; //numarul de descompuneri ale lui n
17     for( int i=0; i<n; ++i )
18     {
19         nrDescompuneri += s[n-1][i] ;
20     }
21     return nrDescompuneri ;
22 }

```

Putem determina o relație de recurență pentru  $S(i, j)$  în cazul general ( $i > j$ ) făcând următoarea observație simplă:

Fiecare dintre cei  $j$  termeni care însumați dau  $i$  sunt mai mari sau egali cu 1. Dacă scădem valoarea 1 din fiecare dintre termeni, suma se va micșora cu  $j$ , iar termenii care fuseseră 1 devin 0, deci dispar.

Ținând cont de această observație și de faptul că numărul de termeni care sunt mai mari strict decât 1 este de minim 1 și maxim  $j$ , putem scrie următoarea relație de recurență:

$$S(i, j) = \sum_{k=1}^j S(i - j, k)$$

Metoda `descompuneNumar()` din **Listing 14.8** rezolvă recurența de mai sus folosind trei cicluri `for` imbricate.

Analizând structura ciclurilor imbricate din metoda de mai sus, reiese că aceasta are o complexitate în  $O(n^3)$ . Putem însă reduce complexitatea la  $O(n^2)$

folosind următorul artificiu de calcul:

$$\begin{aligned} S(i+1, j+1) &= \sum_{k=1}^{j+1} S(i-j, k) = \sum_{k=1}^j S(i-j, k) + S(i-j, j+1) = \\ &= S(i, j) + S(i-j, j+1) \end{aligned}$$

Astfel, termenii  $S(i, j)$  sunt calculați în  $O(1)$  și nu în  $O(n)$ , cum erau calculați înainte. Modificarea algoritmului pentru noua formulă este simplă și o lăsăm ca exercițiu.

În concluzie, deși problemele prezentate în această secțiune nu sunt probleme de programare dinamică propriu-zise, deoarece nu sunt de optimizare, ele constituie un excelent exercițiu pentru înțelegerea principiilor programării dinamice, și mai ales pentru deprinderea abilității de a găsi relații de recurență adecvate pentru exprimarea soluției.

### 14.3 Fundamentare teoretică

În paragraful anterior am făcut o scurtă trecere în revistă a unor probleme simple a căror rezolvare constă în determinarea și calcularea unor recurențe matematice. Am accentuat tratarea iterativă în contrast cu cea recursivă a relațiilor de recurență, subliniind în același timp și modul în care problemele se suprapun în cazul rezolvării recursive. Totuși, aceste probleme nu pun în evidență decât un anumit aspect al programării dinamice, fără a aparține propriu-zis acestui domeniu. În acest capitol vom urmări să vedem care sunt caracteristicile esențiale ale problemelor de programare dinamică pe baza unui exemplu clasic: *problema triunghiului de numere*.

În cazul acestei probleme se dă un triunghi de numere  $z_{ij}$ ,  $j \leq i$  având forma:

Figura 14.4: Triunghi de numere de dimensiune  $n$

```

z11
z21 z22
...
zn1 zn2 ... znn

```

Pentru  $n = 4$ , un posibil triunghi de numere este:

```

2
6 9
3 4 5
9 5 7 6

```

Un *drum* în acest triunghi este o secvență de  $n$  elemente ale triunghiului ( $n$  este numărul de linii) care începe întotdeauna cu elementul de pe prima linie,  $z_{11}$ , (2 în exemplul nostru) și coboară linie cu linie până ajunge pe ultima linie a triunghiului. Din elementul  $z_{ij}$  se poate merge fie în  $z_{i+1,j}$ , fie în  $z_{i+1,j+1}$ . Elementele îngroșate din exemplul următor reprezintă un posibil drum:

```

2
6 9
3 4 5
9 5 7 6

```

Problema constă în a determina un drum în triunghi pentru care suma elementelor să fie maximă. În exemplul nostru, un drum de sumă maximă este:

```

2
6 9
3 4 5
9 5 7 6

```

având valoarea  $2 + 9 + 5 + 7 = 23$ .

Înainte de a ne avânta să rezolvăm o problemă folosind programarea dinamică, trebuie să ne convingem că nu putem folosi și strategii mai simple (cum ar fi Greedy) pentru a obține soluția optimă. O strategie Greedy în cazul problemei triunghiului ar porni din vârful triunghiului și ar coborî la fiecare pas prin elementul mai mare. Aplicând această strategie pe exemplul anterior, am porni din 2, apoi am coborî în 9 (deoarece  $9 > 6$ ), apoi am continua cu 5 respectiv 7. Am obținut astfel drumul 2, 9, 5, 7, care este chiar drumul optim! Am putea fi tentați să credem că putem aplica strategia Greedy. Să considerăm însă exemplul următor:

```

2
4 3
6 5 9

```

Soluția obținută prin strategia Greedy este 2, 4, 6, având lungimea totală 12, în timp ce soluția optimă este 2, 3, 9, având lungimea totală 14. Rezultă așadar că strategia Greedy nu generează întotdeauna soluția optimă.

Pentru a rezolva o problemă printr-un algoritm de programare dinamică, o primă etapă esențială constă în a **descompune (structura) problema în subprobleme asemănătoare ca structură cu problema inițială**.

Structurarea pe subprobleme asemănătoare se face în cazul problemei triunghiului pe baza următoarei observații: fiecare element din triunghi poate fi

considerat vârful unui “sub-triunghi”. De exemplu elementul 9 de pe poziția (2, 2) din triunghiul de mai sus este vârful sub-triunghiului

```

9
4 5
5 7 6

```

iar elementul 6 de pe poziția (2, 1) este vârful sub-triunghiului

```

6
3 4
9 5 7

```

Odată determinate subproblemele, începem prin a le rezolva subproblemele banale (de dimensiune minimă). Astfel, vom începe prin a determina un drum de sumă maximă pentru subproblemele generate de elementele din ultima linie a triunghiului. Acest lucru este banal, întrucât triunghiurile generate de către elementele de pe ultima linie au un singur element, deci drumul maxim este dat chiar de elementele respective:

```

9 5 7 6

```

Având rezolvate subproblemele generate de elementele de pe ultima linie, vom trece să rezolvăm subproblemele generate de elementele de pe penultima linie ( $n-1$ ). Pentru fiecare element de pe penultima linie avem două posibilități: fie se merge pe elementul de pe linia  $n$  aflat chiar dedesubt, fie pe elementul de pe linia  $n$  aflat în dreapta. Evident, vom alege varianta (subproblema) pentru care drumul corespunzător este mai mare. De exemplu, pentru elementul 3, vom prefera să mergem pe traseul  $3 \rightarrow 9$  ( $3 + 9 = 12$ ) decât pe traseul  $3 \rightarrow 5$  ( $3 + 5 = 8$ ). Astfel, valoarea drumului optim pentru subproblemele generate de elementele de pe penultima linie este:

```

12 11 12
9 8 7 6

```

Analog procedăm și pentru elementele aflate pe a antepenultima linie (a doua în cazul nostru) linie. Și în acest caz putem merge fie pe elementul aflat dedesubt fie pe elementul aflat în dreapta-jos. Vom alege, desigur, elementul a cărui subproblemă are valoarea mai mare. De exemplu, din elementul 6 am putea coborî fie prin 3, fie prin 4. Întrucât subproblema generată de 3 are drumul maxim de valoare 12, iar subproblema generată de 4 are drumul maxim de valoare 11, vom alege drumul care trece prin 3. Astfel, valoarea drumului optim pentru subproblemele generate de elementele de pe antepenultima linie este:

```

18 21
12 11 12
9 8 7 6

```

În final, pentru elementul din vârful triunghiului, vom alege să mergem pe traseul din dreapta, deoarece subproblema generată de elementul din dreapta-

jos (9) are valoare mai mare. Obținem astfel valoarea drumului optim pentru elementul din vârf (care este de fapt valoarea optimă pentru problema originală), ca fiind  $2 + 21 = 23$ :

```

23
18 21
12 11 12
9  8  7  6

```

## 14.4 Principiul optimalității

Desigur că structurarea în subprobleme ar fi fost lipsită de sens dacă nu ne-ar fi permis să rezolvăm subproblemele corespunzătoare unui nivel bazându-ne pe soluțiile subproblemelor deja rezolvate. Am anticipat deci faptul că soluția unei subprobleme se determină pe baza soluției subproblemelor de dimensiune mai mică, deja rezolvate. În această situație, vom spune că soluția subproblemei  $x$  *provine* din soluțiile subproblemelor  $y_1, y_2, \dots, y_n$ .

Problemele de programare dinamică sunt în general probleme de *optimizare*, iar problema triunghiului nu face nici ea excepție. Din acest motiv, este necesar ca și subproblemele în care am descompus problema originală să fie tot de optimizare. În fine, cea mai importantă caracteristică a problemelor de programare dinamică este următoarea:

Dacă soluția subproblemei  $x$  *provine* din soluțiile subproblemelor  $y_1, y_2, \dots, y_n$  și soluția subproblemei  $x$  este optimă, atunci și soluțiile subproblemelor  $y_1, y_2, \dots, y_n$  sunt optimale.

Această proprietate poartă numele de **principiul optimalității** și constituie piatra de temelie a programării dinamice.

În general, după ce am realizat o structurare a problemei inițiale în subprobleme, trebuie verificat principiul optimalității. Această verificare se face cel mai adesea prin reducere la absurd.

**Observație:** Dificultatea nu constă în a verifica principiul optimalității, ci în a găsi o structurare a problemei în subprobleme care să verifice acest principiu.

Să demonstrăm acum faptul că problema triunghiului de numere respectă principiul optimalității. O soluție a unei subprobleme constă într-un traseu optimal care pornește din vârful triunghiului corespunzător subproblemei și ajunge la baza triunghiului. Notăm acest traseu cu  $a_k, a_{k+1}, \dots, a_n$ , unde  $a_k$  este un element de pe linia  $k$ ,  $a_{k+1}$  este un element de pe linia  $k + 1$ , iar  $a_n$  este un element de pe ultima linie. Pentru a se respecta principiul optimalității trebuie ca și subtraseul  $a_{k+1}, \dots, a_n$  să fie optimal pentru sub-triunghiul

generat de elementul  $a_{k+1}$ . Verificarea se face ușor prin reducere la absurd. Dacă admitem că ar exista un alt traseu care să pornească din  $a_{k+1}$ , notat cu  $a_{k+1}, b_{k+2}, \dots, b_n$ , care să fie mai bun decât traseul  $a_{k+1}, \dots, a_n$ , atunci traseul  $a_k, a_{k+1}, b_{k+2}, \dots, b_n$  ar fi mai bun decât  $a_k, a_{k+1}, \dots, a_n$ , ceea ce contrazice ipoteza că  $a_k, a_{k+1}, \dots, a_n$  este optim. Rezultă deci că structurarea în subprobleme realizată de noi respectă principiul optimalității.

Nu ne rămâne acum decât să scriem formula de recurență care descrie cum se obține soluția unei subprobleme pe baza subproblemelor de dimensiune mai mică deja rezolvate. Să notăm cu  $l_{ij}$  valoarea traseului optim pentru subproblema generată de elementul  $z_{ij}$ , aflat pe linia  $i$  și coloana  $j$  ( $i \geq j$ ). Este ușor de observat că  $l_{ni} = z_{ni} \forall i = 1 \dots n$ , deoarece drumul maxim pentru elementele de pe ultima linie este reprezentat chiar de elementul respectiv. Am văzut că  $l_{ij}$  se obține fie din  $l_{i+1,j}$  fie din  $l_{i+1,j+1}$  la care se adaugă valoarea elementului  $z_{ij}$ :

$$l_{ij} = z_{ij} + \max\{l_{i+1,j}, l_{i+1,j+1}\} \quad \forall i = 1 \dots n-1, j = 1 \dots i.$$

Valoarea lui  $l_{11}$  reprezintă chiar soluția pentru problema originală.

Calculul recurenței de mai sus este realizat de metoda `drumMaximTriunghi()` din **Listing 14.9**.

### Reciproca principiului optimalității nu este adevărată

O altă observație importantă este aceea că, în general, reciproca principiului optimalității nu este adevărată. Cu alte cuvinte, combinând două sau mai multe sub-soluții optime nu se obține neapărat tot o soluție optimă. Să presupunem că dorim să aflăm drumul de lungime minimă dintre două localități, să zicem Brașov și Constanța. Conform principiului optimalității, dacă avem un drum optim,  $x$ , între Brașov și Constanța care trece prin Ploiești, atunci subdrumul lui  $x$  de la Ploiești la Constanța este de asemenea optim. Totuși, dacă avem un drum optim între Brașov și Suceava și un alt drum optim între Suceava și Constanța, prin combinarea acestor două drumuri vom obține un traseu Brașov-Suceava-Constanța care este departe de a fi optim. Rezultă deci că reciproca principiului optimalității nu este adevărată în acest caz.

Este important să înțelegem că principiul optimalității nu ne asigură că prin combinarea soluțiilor optime se obține tot o soluție optimă. În schimb, el ne asigură că *în căutarea unei soluții optime nu trebuie să luăm în considerare decât subsoluțiile optime*.

Listing 14.9: Rezolvarea recurenței pentru problema triunghiului

```

1 /**
2  * Calculeaza formula de recurenta pentru matricea L.
3  * @return Lungimea drumului maximal pentru triunghi
4  * care se presupun a fi membri ai clasei care contine metoda.
5  */
6 public static int drumMaximTriunghi ()
7 {
8     //initializeaza ultima linie din L
9     for (int i = 0; i < z.length; ++i)
10    {
11        l[z.length - 1][i] = z[z.length - 1][i];
12    }
13
14    //calculeaza elementele din l linie cu linie
15    for (int i = z.length - 2; i >= 0; --i)
16    {
17        for (int j = 0; j <= i; ++j)
18        {
19            l[i][j] = z[i][j] + Math.max(l[i+1][j], l[i+1][j+1]);
20        }
21    }
22
23    return l[0][0];
24 }

```

### O altă descompunere în subprobleme

Este interesant de remarcat faptul că problema triunghiului de numere admite și o altă descompunere în subprobleme care verifică și ea principiul optimalității, și care conduce la o altă soluție a problemei. În descompunerea în subprobleme pe care am dat-o anterior am notat prin  $l_{ij}$  lungimea drumului maxim care începe cu elementul  $z_{ij}$ . De data aceasta, vom nota cu  $m_{ij}$  lungimea drumului maxim care se încheie cu elementul  $z_{ij}$ . Considerând același triunghi de numere:

```

2
6 9
3 4 5
9 5 7 6

```

subproblema corespunzătoare elementului 4 din linia a treia este dată de “tăietura” în sus generată de acest element:

**2**  
**6 9**  
**3 4**

unde drumul maxim care se termină cu 4 a fost îngroșat. Verificarea principiului optimalității se face foarte ușor și de această dată: este clar că un drum de lungime maximă este format din sub-drumuri care sunt tot de lungime maximă.

Odată determinate subproblemele, începem din nou prin a le rezolva pe cele triviale. Astfel, vom determina mai întâi un drum de sumă maximă pentru elementul din prima linie a triunghiului. Întrucât triunghiul generat de către elementul din prima linie are un singur element, drumul optim este dat chiar de elementul respectiv.

Având rezolvate subproblemele generate de elementele de pe prima linie, vom trece să rezolvăm subproblemele generate de elementele de pe următoarele linii, obținând triunghiul:

**2**  
**8 11**  
**11 15 16**  
**20 20 23 22**

Pentru fiecare element al triunghiului (mai puțin cele din prima și ultima coloană a fiecărei linii) avem două posibilități: fie se ajunge la el de pe elementul aflat imediat deasupra, fie de pe elementul aflat pe diagonală stânga-sus. Evident, vom alege traseul (subproblema) pentru care drumul corespunzător este mai mare. De exemplu, pentru elementul 4, vom prefera să mergem pe traseul  $\dots \rightarrow 9 \rightarrow 4$  (având lungimea  $11 + 4 = 15$ ) decât pe traseul  $\dots \rightarrow 6 \rightarrow 4$  (având lungimea  $8 + 4 = 12$ ).

Soluția subproblemei inițiale este dată de subproblema corespunzătoare elementului de pe ultima linie, a cărei valoare este maximă. Rezultă deci că lungimea drumului maxim este 23, așadar am regăsit valoarea obținută aplicând cealaltă descompunere în subprobleme. Formulele de recurență pentru calculul elementelor  $m_{ij}$  sunt:

$$m_{ij} = z_{ij} + \max(m_{i-1, j-1}, m_{i-1, j}) \quad m_{11} = z_{11},$$

unde facem convenția că  $m_{0i} = m_{i+1i} = 0$ .

#### 14.4.1 Metoda “înainte” și metoda “înapoi”

Așa cum probabil ați remarcat deja, cele două descompuneri în subprobleme pe care le-am descris pentru problema triunghiului de numere sunt complementare:



- în prima variantă se consideră sub-drumuri care încep cu un anumit element, în cea de-a doua variantă se consideră sub-drumuri care se termină cu un anumit element;
- în prima variantă se construiește drumul optim pornind de la baza triunghiului, în timp ce în cea de-a doua variantă se construiește drumul optim pornind de la vârful triunghiului;
- în prima variantă recurența merge “înainte” de la elementul  $i$  la elementul  $i + 1$ , în cea de-a doua variantă recurența merge “înapoi”, de la elementul  $i$  la elementul  $i - 1$ .

Unii autori clasifică metodele de rezolvare a problemelor de programare dinamică în trei clase:

1. Metoda “înainte”, în care ne folosim de faptul că optimalitatea subsoluției  $a_k, a_{k+1}, \dots, a_n$  implică optimalitatea subsoluției  $a_{k+1}, \dots, a_n$ ;
2. Metoda “înapoi”, în care ne folosim de faptul că optimalitatea subsoluției  $a_1, \dots, a_{k-1}, a_k$  implică optimalitatea subsoluției  $a_1, \dots, a_{k-1}$ ;
3. Metoda “mixtă”, în care ne folosim de faptul că optimalitatea soluției de tipul  $a_1, \dots, a_k, \dots, a_n$  implică optimalitatea atât a subsoluției  $a_1, \dots, a_{k-1}$  cât și a sub-soluției  $a_{k+1}, \dots, a_n$ .

În prima descompunere în subprobleme am aplicat metoda “înainte”, iar în a doua descompunere în subprobleme am aplicat metoda “înapoi”. Vom prezenta în paragraful 14.5 și un exemplu de problemă (parantezarea unui șir de matrice) în care se aplică metoda “mixtă”.

### 14.4.2 Determinarea efectivă a soluției optime

Deși rezolvarea recurențelor asociate unei probleme de programare dinamică ne conduce la aflarea valorii optime căutate, ea nu furnizează și soluția efectivă pentru care se obține acea valoare optimă. De exemplu, în cazul problemei triunghiului de numere am calculat care este valoarea drumului optimal, dar nu am găsit efectiv care este drumul pentru care se obține valoarea optimă.

Vestea bună este că aflarea soluției optime a problemei este în general simplă și se bazează de cele mai multe ori pe reconstituirea “traseului” prin care s-a obținut valoarea soluției optime. În cazul problemei triunghiului de numere putem afla cu ușurință care este drumul pentru care se obține soluția optimă,

reconstituind traseul prin care s-a obținut valoarea optimă. Pentru exemplul nostru, triunghiul construit prin rezolvarea relațiilor de recurență era:

```

23
18  21
12  11  12
9   5   7  6

```

Se observă cu ușurință că valoarea 23 din vârful triunghiului a fost obținută din elementul 21 aflat în dreapta jos (prin adăugarea valorii 2 din vârful triunghiului original), valoarea 21 a fost obținută din elementul 12 aflat la dreapta, iar 12 din 7.

Metoda `afiseazaDrumMaxim()` din **Listing 14.10** calculează drumul maxim pentru problema triunghiului. La linia 33 se inițializează variabila `s` cu valoarea drumului maxim, aflată în vârful triunghiului. Drumul propriu-zis este construit în ciclul `while` din liniile 38-53, în care se parcurge matricea `l` de sus în jos, linie cu linie. La fiecare iterație, variabila `s` conține valoarea către care trebuie să coborâm. Astfel, dacă `s` este egală cu `l[i+1][j+1]` (linia 46) se coboară direct în `l[i+1][j+1]`, altfel se coboară în `l[i+1][j+2]`.

Listing 14.10: Soluția completă a problemei drumului maxim

```

1 import java.io.*;
2 import io.Reader;
3
4 public class DrumulMaxim
5 {
6     public static int drumMaximTriunghi(int[][] z)
7     {
8         int[][] l = new int[z.length][z.length];
9
10        for (int i = 0; i < z.length; i++)
11        {
12            l[z.length - 1][i] = z[z.length - 1][i];
13        }
14
15        for (int i = z.length - 2; i >= 0; i--)
16        {
17            for (int j = 0; j <= i; j++)
18            {
19                l[i][j] = z[i][j] +
20                    Math.max(l[i + 1][j],
21                        l[i + 1][j + 1]);
22            }
23        }
24
25        afiseazaDrumMaxim(z, l);
26
27        return l[0][0];

```

```

28 }
29
30 public static void afiseazaDrumMaxim(int[][] z, int[][] l)
31 {
32     int s = l[0][0];
33     int i = 0;
34     int j = 0;
35
36     System.out.print("Drumul maxim: ");
37     while (s != 0)
38     {
39         System.out.print(z[i][j] + " ");
40
41         s -= z[i][j];
42
43         if (i + 1 < z.length)
44         {
45             if (s == l[i + 1][j + 1])
46             {
47                 j++;
48             }
49
50             i++;
51         }
52     }
53     System.out.println();
54 }
55
56 public static void main(String args[])
57 {
58     System.out.print("Dimensiune triunghi: ");
59     int n = Reader.readInt();
60
61     int[][] z = new int[n][n];
62     System.out.println("Elementele triunghiului:");
63     for (int i = 0; i < n; i++)
64     {
65         for (int j = 0; j <= i; j++)
66         {
67             System.out.print("z[" + i + "][" + j
68                 + "] = ");
69
70             z[i][j] = Reader.readInt();
71         }
72     }
73
74     System.out.println("Suma maxima = " + drumMaximTriunghi(z));
75 }
76 }

```

## 14.5 Înmulțirea unui șir de matrice

Următorul exemplu de problemă de programare dinamică pe care îl vom studia este problema înmulțirii optimale a unui șir de matrice. În cazul acestei probleme se dă o secvență  $A_1, A_2, \dots, A_n$  de matrice care trebuie înmulțite. Cu alte cuvinte, se dorește calcularea produsului

$$A_1 A_2 \dots A_n,$$

unde  $A_i$  are dimensiunile  $d_{i-1} \times d_i$ . Având în vedere faptul că înmulțirea matricelor este asociativă, calculul produsului de mai sus se poate face în mai multe moduri, în funcție de ordinea în care decidem să realizăm operațiile de înmulțire. De exemplu, pentru  $n = 3$  există două moduri în care se poate calcula produsul:

$$A_1 (A_2 A_3)$$

sau

$$(A_1 A_2) A_3.$$

Unii cititori își pot pune în mod legitim întrebarea: *Ținând cont de faptul că înmulțirea matricelor este asociativă, oricum am pune parantezele, rezultatul final al calculului va fi același. Așadar ce sens are să ne preocupăm de ordinea de realizare a operațiilor de înmulțire?* Răspunsul este că numărul de operații elementare de înmulțire este diferit, funcție de modul în care alegem să punem parantezele. De exemplu, pentru  $n = 3$ , să presupunem că cele 3 matrice au respectiv dimensiunile  $(10, 50)$ ,  $(50, 20)$  și  $(20, 1)$ . Numărul de operații necesar pentru a înmulți două matrice de dimensiune  $(m, n)$  și  $(n, p)$  este  $m \cdot n \cdot p$ , după cum reiese clar din algoritmul din **Listing 14.11** (considerăm ca barometru operația de înmulțire scalară  $a[i][k] * b[k][j]$ ). Dacă vom calcula produsul celor 3 matrice după primul mod  $(A_1 (A_2 A_3))$ , vom face  $50 \times 20 \times 1 = 1000$  de operații pentru a calcula produsul  $A_2 A_3$ , plus încă  $10 \times 50 \times 1 = 500$  de operații pentru a înmulți rezultatul cu  $A_1$ . Așadar, în total vom realiza  $1000 + 500 = 1500$  de înmulțiri scalare. Dacă vom calcula produsul celor 3 matrice în al doilea mod  $((A_1 A_2) A_3)$ , vom face  $10 \times 50 \times 20 = 10000$  de operații pentru a calcula produsul  $A_1 A_2$  plus încă  $10 \times 20 \times 1 = 400$  operații pentru a calcula produsul rezultatului cu  $A_3$ . Vom face așadar un total de 10400 de operații ceea ce înseamnă cam de 7 ori mai mult decât în varianta precedentă. În concluzie, are sens să ne punem problema de a găsi cea mai eficientă metodă de a realiza această înmulțire.

O soluție imediată a problemei ar fi să se găsească toate modurile posibile de parantezare a șirului de matrice și să se aleagă cea pentru care numărul de

operații este minim. Totuși, numărul de parantezări pentru un șir de lungime  $n$  este (vezi [Cormen], p. 261)

$$\frac{1}{n}C_{2n-2}^{n-1} \in \Omega(4^n/n^{3/2})$$

ceea ce exclude din start posibilitatea căutării exhaustive datorită numărului exponențial de variante.

Listing 14.11: Înmulțirea a două matrice de dimensiune  $m \times n$  respectiv  $n \times p$ . Se observă cu ușurință că numărul de operații de înmulțire realizate este  $m \cdot n \cdot p$

```

1 /**
2  * Calculeaza produsul a doua matrice de numere intregi.
3  * @return O matrice de numere intregi care reprezinta produsul
4  * matricelor date ca parametru.
5  */
6 public static int[][] produs(int[][] a, int[][] b)
7 {
8     int[][] produs = new int[a.length][b[0].length];
9     for (int i = 0; i < a.length; ++i)
10     {
11         for (int j = 0; j < b[0].length; ++j)
12         {
13             produs[i][j] = 0;
14             for (int k = 0; k < b.length; ++k)
15             {
16                 produs[i][j] += a[i][k]*b[k][j];
17             }
18         }
19     }
20
21     return produs;
22 }

```

## Caracterizarea substructurii optime

O parantezare optimă a produsului  $A_1 A_2 \dots A_n$  împarte secvența între  $A_k$  și  $A_{k+1}$ , unde  $k$  este un număr natural în intervalul  $1..n-1$ . Aceasta înseamnă că mai întâi se calculează  $A_1 \dots A_k$ , apoi  $A_{k+1} \dots A_n$  și în final se înmulțesc cele două rezultate pentru a obține produsul  $A_1 A_2 \dots A_n$ . Numărul total de înmulțiri realizate este egal cu numărul de înmulțiri necesare pentru calculul lui  $A_1 \dots A_k$  plus numărul de înmulțiri necesare pentru a calcula  $A_{k+1} \dots A_n$  la care se adaugă numărul de operații necesar pentru a înmulți cele două rezultate.

Această modalitate de împărțire ne duce cu gândul la a considera subprobleme de forma  $P_{ij} = A_i A_{i+1} \dots A_j$ , cu  $1 \leq i \leq j \leq n$ . Observația esențială în cazul acestei probleme este că dacă parantezarea optimă pentru a calcula  $A_i \dots A_j$  împarte produsul pe poziția  $k$ ,

$$P_{ij} = (A_i \dots A_k)(A_{k+1} \dots A_j)$$

atunci parantezarea subșirului  $A_i \dots A_k$  din cadrul parantezării optime a șirului  $A_i \dots A_j$  este o parantezare optimă pentru șirul  $A_i \dots A_k$  (dacă nu ar fi optimă, atunci înlocuirea respectivei parantezări cu cea optimă ar produce o altă parantezare pentru  $A_i \dots A_j$  al cărei cost ar fi mai mic decât cel optim, ceea ce este absurd). O observație similară este valabilă și pentru parantezarea lui  $A_{k+1} \dots A_j$  din cadrul parantezării optime a lui  $A_i \dots A_j$ , care trebuie să fie optimă pentru șirul  $A_{k+1} \dots A_j$ . Am arătat așadar că o soluție optimă pentru problema noastră este compusă din subsoluții optime ale subproblemelor, ceea ce ne permite să aplicăm programarea dinamică.

### Obținerea formulelor de recurență

Odată găsită o descompunere în subprobleme care respectă principiul optimității, putem trece la definirea valorii unei soluții optime în funcție de soluțiile optime ale subproblemelor. Fie  $m_{ij}$  numărul minim de înmulțiri necesare pentru a calcula  $P_{ij} = A_i \dots A_j$ . Scopul final este să calculăm  $m_{1n}$ . Dacă  $i = j$ , atunci șirul constă într-o singură matrice, deci nu avem nevoie de nici o înmulțire scalară. Vom avea astfel

$$m_{ii} = 0, \forall i = 1..n.$$

Pentru a calcula  $m_{ij}$  în cazul general ( $i < j$ ) ne vom folosi de substructura optimă definită în paragraful anterior. Să presupunem că o parantezare optimă a șirului  $A_i \dots A_j$  împarte produsul între  $A_k$  și  $A_{k+1}$  cu  $i \leq k < j$ . Am arătat deja că în această situație numărul de operații necesar pentru a calcula  $P_{ij}$  este egal cu numărul de operații necesare pentru a calcula  $P_{ik}$  plus numărul de operații necesare pentru a calcula  $P_{k+1j}$  plus costul înmulțirii celor două matrice rezultat. Având în vedere faptul că  $P_{ik}$  are dimensiunea  $(d_{i-1}, d_k)$ , iar  $P_{k+1j}$  are dimensiunea  $(d_k, d_j)$ , evaluarea produsului  $P_{ik}P_{k+1j}$  necesită  $d_{i-1}d_kd_j$  operații. Obținem astfel numărul de operații necesare pentru a calcula  $m_{ij}$ :

$$m_{ij} = m_{ik} + m_{k+1j} + d_{i-1}d_kd_j.$$

Ecuția de mai sus este valabilă în situația în care cunoaștem poziția  $k$  în care se descompune produsul. Cum noi nu știm care este această poziție, vom încerca

toate variantele posibile pentru  $k$  și o vom alege pe cea pentru care se obține valoarea minimă. Din fericire, există doar  $j - i$  valori posibile pentru  $k$ , și anume  $k = i, i + 1, \dots, j - 1$ . În consecință, formula recurentă pentru definirea costului minim a parantezării produsului  $A_i \dots A_j$  devine

$$m_{ij} = \begin{cases} 0 & \text{daca } i = j \\ \min_{i \leq k < j} \{m_{ik} + m_{k+1j} + d_{i-1}d_kd_j\} & \text{daca } i < j \end{cases} \quad (14.1)$$

### Rezolvarea recurenței și calculul costului optimal

În acest moment este ușor să scriem un algoritm care să calculeze recurența pentru  $m_{ij}$ . O primă alternativă, destul de tentantă, este aceea de a implementa calculul recurenței direct, folosind un algoritm recursiv. O analiză similară cu cea de la calculul recursiv al șirului lui Fibonacci (paragraful 14.2.1), ne convinge că această modalitate de calcul generează o complexitate exponențială, cu nimic mai bună decât încercarea tuturor parantezărilor posibile.

În locul calculului recursiv al lui  $m_{ij}$  vom folosi din nou un tabel în care vom reține valorile intermediare ale subproblemelor. Ca de obicei, calculul se va face *bottom-up*, adică se rezolvă mai întâi subproblemele de dimensiune mică, urmate apoi succesiv de subprobleme de dimensiune mai mare, până când se ajunge la problema inițială. Metoda `sirMatrice()` din **Listing 14.12** implementează această strategie.

`sirMatrice()` calculează matricea  $m$  mergând pe probleme de dimensiune din ce în ce mai mare. La prima iterație ( $l = 0$ ) algoritmul calculează  $m[i][i + 1]$  (costul șirurilor de lungime 2) pentru  $i = 0 \dots n - 2$  (indicii sunt deplasați față de notația anterioară deoarece în Java indexarea tablourilor începe de la 0). La a doua trecere prin ciclu ( $l = 1$ ) se calculează  $m[i][i + 2]$  pentru  $i = 0, 1, \dots, n - 3$  (costul șirurilor de lungime 3) etc. Așadar, elementele matricei  $m$  sunt calculate pe diagonală, mergând paralel cu diagonala principală, până când se ajunge în colțul din dreapta sus al matricei. **Figura 14.5** ilustrează această procedură pentru un șir de  $n = 6$  matrice, având dimensiunile  $(20 \times 15), (15 \times 30), (30 \times 5), (5 \times 25), (25 \times 10), (10 \times 35)$ .

### Construirea efectivă a unei soluții optime

Și în cazul acestei probleme, rezolvarea recurenței implică găsirea valorii optime, fără a da însă și parantezarea efectivă pentru care s-a obținut valoarea optimă. Ca de obicei, calculul soluției optime se face simplu, folosindu-ne de rezolvarea recurenței. Pentru a putea găsi care este soluția optimă, este suficient ca pentru fiecare produs  $A_i \dots A_j$  să știm care este poziția  $k$  la care se împarte

Listing 14.12: Calculul recurenței pentru înmulțirea șirului de matrice folosind un tabel  $m$  în care se rețin valorile subproblemelor

```

1 /**
2  * Calculeaza formula de recurenta pentru matricea m.
3  * Ne folosim de faptul ca elementele matricei l sunt
4  * initializate implicit cu 0.
5  * @return Costul optim pentru inmultirea matricelor a caror
6  * dimensiune este retinuta in sirul d care se considera a
7  * fi membru al clasei care contine metoda.
8  */
9 public static int sirMatrice()
10 {
11     int n = d.length - 1;
12     int [][] m = new int[n][n];
13     for (int l = 0; l < n - 1; ++l) //l este lungimea subsirului
14     {
15         for (int i = 0; i < n - l - 1; ++i)
16         {
17             j = i + l + 1;
18             m[i][j] = Integer.MAX_VALUE;
19             int temp;
20
21             for (int k = i; k < j; ++k)
22             {
23                 temp = m[i][k] + m[k+1][j] + d[i]*d[k+1]*d[j+1];
24                 if (temp < m[i][j])
25                 {
26                     m[i][j] = temp;
27                 }
28             }
29         }
30     }
31
32     return m[0][n - 1];
33 }

```

în două subprocese. Aceasta implică să construim o matrice  $poz_{ij}$  care reține poziția în care șirul  $A_i \dots A_j$  este împărțit în subprocese. Calculul lui  $poz_{ij}$  se face simplu, adăugând atribuirea

$$poz[i][j] = k;$$

după linia 26 din metoda `sirMatrice()` (**Listing 14.12**). Având la dispoziție șirul  $poz$ , găsirea soluției optime se face simplu folosind metoda recursivă `sirMatriceSolutie()` din **Listing 14.13**.



Figura 14.5: Tabelul  $m$  rezultat în urma rezolvării recurenței pentru problema înmulțirii șirului de matrice, pentru un șir de 6 matrice având dimensiunile  $(20 \times 15)$ ,  $(15 \times 30)$ ,  $(30 \times 5)$ ,  $(5 \times 25)$ ,  $(25 \times 10)$ ,  $(10 \times 35)$

0	9000	3750	6250	6000	13000
	0	2250	5625	7250	12500
		0	3750	2750	13250
			0	1250	10500
				0	8750
					0

Listing 14.13: Construirea efectivă a soluției optime pentru problema înmulțirii șirului de matrice

```

1 /**
2  * Metoda care determina efectiv produsul sirului de
3  * matrice in mod optim bazandu-se pe matricea calculata
4  * de metoda sirMatrice.
5  * i si j reprezinta limitele intre care se calculeaza produsul
6  * @return String ce reprezinta modul de parantezare al matricelor
7  */
8 public static String sirMatriceSolutie(int[][] poz, int i, int j)
9 {
10     if (i < j) //sirul contine cel putin doua matrice
11     {
12         String s1 = sirMatriceSolutie(poz, i, poz[i][j]);
13         String s2 = sirMatriceSolutie(poz, poz[i][j]+1, j);
14
15         return "(" + s1 + "*" + s2 + ")";
16     }
17 }
18 else //sirul contine o singura matrice
19 {
20     return "A" + String.valueOf(i);
21 }
22 }

```

Clasa `Matrice` din **Listing 14.14** afișează parantezarea optimă și numărul de înmulțiri minim necesar pentru un șir de matrice ale cărui dimensiuni sunt citite de la tastatură.

Listing 14.14: Soluția completă a problemei șirului de matrice

```

import java.io.*;

```

```

2 import io.Reader;
3
4 public class Matrice
5 {
6     public static int sirMatrice(int[] d)
7     {
8         int n = d.length - 1;
9         int[][] m = new int[n][n];
10        int[][] poz = new int[n][n];
11
12        for (int l = 0; l < n - 1; l++)
13        {
14            for (int i = 0; i < n - l - 1; i++)
15            {
16                int j = i + l + 1;
17                m[i][j] = Integer.MAX_VALUE;
18                int temp;
19                for (int k = i; k < j; k++)
20                {
21                    temp = m[i][k] + m[k + 1][j] +
22                        d[i] * d[k + 1] * d[j + 1];
23
24                    if (temp < m[i][j])
25                    {
26                        m[i][j] = temp;
27                        poz[i][j] = k;
28                    }
29                }
30            }
31        }
32
33        System.out.print("Ordinea inmultirii matricilor: " +
34            sirMatriceSolutie(poz, 0, n - 1));
35        System.out.println();
36
37        return m[0][n - 1];
38    }
39
40    public static String sirMatriceSolutie(int[][] poz,
41        int i, int j)
42    {
43        if (i < j)
44        {
45            String s1 = sirMatriceSolutie(poz, i, poz[i][j]);
46            String s2 = sirMatriceSolutie(poz, poz[i][j] + 1, j);
47
48            return "(" + s1 + " * " + s2 + ")";
49        }
50        else

```

```

52     {
53         return "A" + String.valueOf(i);
54     }
55 }
56
57 public static void main(String[] args)
58 {
59     System.out.print("Numarul de matrice: ");
60     int n = Reader.readInt();
61
62     if (n < 1) return;
63
64     int[] d = new int[n + 1];
65
66     System.out.println("Sirul dimensiunilor: ");
67
68     for (int i = 0; i < n + 1; i++)
69     {
70         System.out.print("d[" + i + "]=");
71         d[i] = Reader.readInt();
72     }
73
74     System.out.println("Numarul minim de operatii necesar " +
75         "inmultirii matricilor: " + sirMatrice(d));
76 }
77 }

```

## 14.6 Subșir comun de lungime maximă

Următoarea problemă clasică de programare dinamică pe care o vom analiza este problema subșirului comun de lungime maximă. Un subșir al unui șir  $x = x_1x_2 \dots x_n$  se obține prin eliminarea din șirul inițial  $x$  a unor componente  $x_{i_1}, x_{i_2}, \dots, x_{i_k}$ ; componentele care se elimină nu trebuie să fie neapărat consecutive. De exemplu, șirul  $aeg$  este un subșir al lui  $abcdefgh$ . Astfel, dându-se două șiruri,  $x$  și  $y$ , spunem că șirul  $z$  este un *subșir comun* pentru  $x$  și  $y$  dacă  $z$  este un subșir atât pentru  $x$  cât și pentru  $y$ .

În problema subșirului comun de lungime maximă se dau două șiruri  $x = x_1x_2 \dots x_m$  și  $y = y_1y_2 \dots y_n$  arbitrare și se cere să se găsească un subșir *comun* al lor  $z = z_1z_2 \dots z_p$  (evident,  $p \leq \min(m, n)$ ) care să aibă lungimea maximă.

**Exemplu:** Pentru șirurile *dinamic* și *inadecvat* subșirul comun de lungime maximă este *inac*, deoarece nu există nici un subșir comun de lungime mai mare decât 4. Este important de observat că subșirul comun de lungime maximă nu este neapărat unic. Astfel, pentru șirurile *abeiro* și *bueor* un subșir

comun de lungime maximă este *ber*, iar altul este *beo*.

O abordare brutală a acestei probleme constă în generarea tuturor subșirurilor lui  $x$  și alegerea celui mai lung care este în același timp și subșir al lui  $y$ . Desigur, această soluție nu este deloc eficientă, având în vedere faptul că există  $2^m$  subșiruri ale lui  $x$ , deci timpul necesar pentru rezolvarea problemei ar fi exponențial.

### Caracterizarea substructurii optime (verificarea principiului optimalității)

Din fericire, această problemă se pretează la o structurare pe subprobleme care verifică principiul optimalității. Totuși, în acest caz, structurarea în subprobleme care să verifice principiul optimalității nu este evidentă.

Subproblemele se definesc în mod natural pe baza unor perechi de “prefixe” ale celor două șiruri de intrare. Mai exact, dându-se un șir  $x = x_1x_2 \dots x_m$ , un prefix al său este șirul  $x_1x_2 \dots x_i$ , cu  $i = 0, 1, \dots, m$ . Astfel, pentru fiecare pereche de prefixe,  $x = x_1 \dots x_i$ ,  $y = y_1 \dots y_j$  ale șirurilor inițiale vom calcula lungimea subșirului maximal comun, ajungând în final să rezolvăm și problema noastră.

Pentru a verifica principiul optimalității vom enunța trei propoziții simple, în care plecăm de la ipoteza că  $z_1z_2 \dots z_p$  este un subșir maximal pentru  $x_1x_2 \dots x_m$  și  $y_1y_2 \dots y_n$ .

1. Dacă  $x_m = y_n$ , atunci  $z_1z_2 \dots z_{p-1}$  este un subșir comun maximal pentru  $x_1x_2 \dots x_{m-1}$  și  $y_1y_2 \dots y_{n-1}$ ;
2. Dacă  $x_m \neq y_n$  și  $z_p \neq x_m$ , atunci  $z_1z_2 \dots z_p$  este un subșir comun maximal pentru  $x_1x_2 \dots x_{m-1}$  și  $y_1y_2 \dots y_n$ ;
3. Dacă  $x_m \neq y_n$  și  $z_p \neq y_n$ , atunci  $z_1z_2 \dots z_p$  este un subșir comun maximal pentru  $x_1x_2 \dots x_m$  și  $y_1y_2 \dots y_{n-1}$ .

Demonstrația celor trei afirmații de mai sus se face în mod banal prin reducere la absurd și o lăsăm ca exercițiu.

Caracterizarea din propozițiile anterioare arată că un subșir comun maximal pentru două șiruri conține un subșir comun maximal pentru prefixele celor două șiruri, deci problema are proprietatea de substructură optimă. Tot propozițiile de mai sus furnizează și formula de recurență care rezolvă problema.

Vom nota cu  $l[i, j]$  lungimea subșirului comun maximal pentru prefixele  $x_1 \dots x_i$  și  $y_1 \dots y_j$ .

Ca și condiții inițiale avem:

$$l[0, j] = 0, \quad \forall j = 0, 1, \dots, n$$

$$l[i, 0] = 0, \quad \forall i = 0, 1, \dots, m$$

deoarece atunci când unul dintre șiruri este vid (are lungime 0), subșirul comun maximal va fi întotdeauna vid.

Din cele trei propoziții rezultă în mod clar faptul că având un subșir comun maximal pentru două șiruri oarecare  $x_1x_2 \dots x_m$  și  $y_1y_2 \dots y_n$ , acesta se va regăsi în subșirul comun maximal pentru una din variantele

$x_1x_2 \dots x_{m-1}$  și  $y_1y_2 \dots y_{n-1}$  dacă  $x_m = y_n$ ,

sau

$x_1x_2 \dots x_{m-1}$  și  $y_1y_2 \dots y_n$  ori  $x_1x_2 \dots x_m$  și  $y_1y_2 \dots y_{n-1}$  dacă  $x_m \neq y_n$ .

De aici reiese că pentru a calcula  $l[m, n]$  sunt necesare valorile  $l[m-1, n-1]$ ,  $l[m-1, n]$  și  $l[m, n-1]$ . Totuși, cele trei propoziții nu dau indicații exacte despre formula de recurență decât într-un singur caz, și anume când  $x_m = y_n$  (noi nu știm ce valoare are  $z_p$  atunci când construim soluția, deci nu putem distinge între variantele 2 și 3). În situația în care  $x_m \neq y_n$  putem alege fie  $l[m-1, n]$ , fie  $l[m, n-1]$ . Dintre aceste valori o vom alege evident pe cea mai mare, deoarece ea îndeplinește proprietatea de optimalitate (se poate arăta acest lucru ușor, prin reducere la absurd). Astfel, vom alege

$$l[m, n] = \max\{l[m-1, n], l[m, n-1]\} \text{ } x_m \neq y_n$$

Așadar, formula de recurență completă este:

$$l[i, j] = \begin{cases} 0 & \text{ptr } i = 0 \text{ sau } j = 0, \\ l[i-1, j-1] + 1 & \text{ptr } i = 1..m, j = 1..n \text{ si } x_i = x_j, \\ \max(l[i-1, j], l[i, j-1]) & \text{ptr } i = 1..m, j = 1..n \text{ si } x_i \neq x_j. \end{cases}$$

Calculul recurenței de mai sus se face simplu, folosind metoda `subsirComunMaximal()` din **Listing 14.15**.

**Listing 14.15:** Rezolvarea recurenței pentru subșirul comun de lungime maximă

```
1 /**
2  * Calculeaza formula de recurenta pentru matricea l.
3  * Ne folosim de faptul ca elementele matricei l sunt
4  * initializate implicit cu 0.
5  * @return Lungimea subisrului comun maximal pentru x si y
6  * care se presupun a fi membri ai clasei care contine metoda.
7  */
8 public static int subsirComunMaximal()
9 {
10     int m = x.length(); //x este String
11     int n = y.length(); //y este String
```

```

12  for (int i = 0; i < m; ++i)
13  {
14      for (int j = 0; j < n; ++j)
15      {
16          if (x.charAt(i) == y.charAt(j))
17          {
18              l[i+1][j+1] = l[i][j] + 1;
19          }
20          else
21          {
22              l[i+1][j+1] = Math.max(l[i][j+1], l[i+1][j]);
23          }
24      }
25  }
26
27  return l[m][n];
28
29 }

```

Am rezolvat astfel problema determinării lungimii subșirului comun maximal, dar, ca de obicei, nu am găsit subșirul efectiv care are lungimea maximă. Determinarea subșirului efectiv se poate realiza ușor pe baza matricei  $l$  calculată în metoda `subsirComunMaximal()` din **Listing 14.15**, urmărind care a fost “traseul” care a condus la subșirul de lungime maximă (dat de  $l[m, n]$ ). Am evidențiat deja faptul că  $l[i, j]$  poate proveni doar dintr-unul din elementele  $l[i-1, j-1]$ ,  $l[i-1, j]$ , sau  $l[i, j-1]$ . Începem cu elementul  $l[m, n]$  al matricei; acesta provine din  $l[m-1, n-1]$  dacă  $x_m = y_n$  sau din maximul dintre  $l[m-1, n]$  și  $l[m, n-1]$  în caz contrar. Mergem astfel pas cu pas până când ajungem la  $l[0, 0]$ . Din drumul de la  $l[m, n]$  la  $l[0, 0]$  construit anterior reținem doar elementele  $l[i, j]$  pentru care  $x_i = y_j$ . Afișând în ordine inversă aceste elemente, obținem chiar șirul cerut (**Listing 14.16**).

Listing 14.16: Determinarea efectivă a subșirului comun maximal

```

1 /**
2  * Determina care este subsirul comun maximal pe baza
3  * matricei l si a sirurilor x si y (care se presupun a fi
4  * membri ai clasei.
5  * @return Un String care reprezinta subsirul comun maximal.
6  */
7 public static String determinaSubsir(int i, int j)
8 {
9     if (i != 0 && j != 0)
10     {
11         if (x.charAt(i - 1) == y.charAt(j - 1))

```

```

12     {
13         return determinaSubsir(i - 1, j - 1) + x.charAt(i - 1);
14     }
15     else
16     {
17         if (l[i][j] == l[i-1][j])
18         {
19             return determinaSubsir(i - 1, j);
20         }
21         else
22         {
23             return determinaSubsir(i, j - 1);
24         }
25     }
26 }
27 else
28 {
29     return "";
30 }
31 }

```

Clasa `SubsirComun` din **Listing 14.17** citește două stringuri de la tastatură, după care afișează subșirul lor comun de lungime maximă.

Listing 14.17: Soluția completă a problemei subșirului comun maximal

```

1 import java.io.*;
2 import io.Reader;
3
4 public class SubsirComun
5 {
6     private static String x;
7     private static String y;
8     private static int [][] l;
9
10    public static int subsirComunMaximal()
11    {
12        // ...
13        System.out.println("Subsirul comun maximal: " +
14            determinaSubsir(m, n));
15
16        return l[m][n];
17    }
18
19    public static String determinaSubsir(int i, int j)
20    {
21        // ...
22    }
23
24    public static void main(String args[])
25    {

```

```

26     System.out.print("x = ");
27     x = Reader.readString();
28
29     System.out.print("y = ");
30     y = Reader.readString();
31
32     System.out.println("Lungimea subsirului comun maximal: "
33         + subsirComunMaximal());
34 }
35 }

```

## 14.7 Distanța Levensthein

O problemă foarte asemănătoare ca structură cu problema subșirului comun de lungime maximă este calculul distanței Levensthein între două șiruri de caractere. Să presupunem că avem două șiruri de caractere,  $x = x_1x_2 \dots x_m$  și  $y = y_1y_2 \dots y_n$ . Operațiile permise asupra unui șir de caractere sunt:

- ștergerea unui caracter oarecare din șir;
- inserarea unui caracter pe o poziție oarecare din șir;
- înlocuirea unui caracter oarecare cu un altul.

Se cere să se determine numărul minim de operații prin care șirul  $x$  se poate transforma în șirul  $y$ . Acest număr se numește *distanța Levensthein* dintre  $x$  și  $y$  și reprezintă o evaluare mult mai obiectivă a similarității a două șiruri decât distanța Hamming (care pur și simplu numără pozițiile pe care cele două șiruri diferă).

**Exemplu:** Pentru a transforma șirul *algorithm* în *aborigen* sunt necesare următoarele operații:

$algorithm[l \leftrightarrow b] \rightarrow abgoritm[sterge\ g] \rightarrow aboritm[t \leftrightarrow g] \rightarrow$   
 $\rightarrow aborigem[inseareaza\ e] \rightarrow aborigem[m \leftrightarrow n] \rightarrow aborigen$

Este ușor de văzut că întotdeauna există o secvență de astfel de operații care să transforme un șir într-altul, deci nu se pune problema de fezabilitate. Structurarea în subprobleme este foarte asemănătoare cu cea de la problema subșirului comun de lungime maximă din secțiunea precedentă. Astfel, pentru oricare prefix  $X_i = x_1 \dots x_i$  al lui  $x$  și oricare prefix  $Y_j = y_1 \dots y_j$  al lui  $y$  ne vom propune să determinăm secvența cea mai scurtă de operații pentru a-l transforma pe  $X_i$  în  $Y_j$ .



Să arătăm mai întâi că descompunerea în subprobleme dată de noi respectă principiul optimalității. Pentru aceasta, va trebui să găsim o modalitate de a construi soluția optimă a unei probleme pe baza soluțiilor optime ale problemelor de dimensiune mai mică. Să considerăm două prefixe  $X_i$  și  $Y_j$  ale șirurilor inițiale, iar  $s = s_1 s_2 \dots s_p$  secvența optimă de operații pentru a-l transforma pe primul în al doilea. Putem face acum următoarele observații:

- Dacă  $x_i = y_j$ , atunci  $s = s_1 \dots s_p$  este o secvență optimă de transformări și pentru  $X_{i-1}, Y_{j-1}$ ;
- Dacă  $x_i \neq y_j$  și ultima transformare din  $s$  este o operație de ștergere, atunci  $s_1 \dots s_{p-1}$  este o secvență optimă de transformări pentru  $X_{i-1}$  și  $Y_j$ ;
- Dacă  $x_i \neq y_j$  și ultima transformare din  $s$  este o operație de inserare, atunci  $s_1 \dots s_{p-1}$  este o secvență optimă de transformări pentru  $X_i$  și  $Y_{j-1}$ ;
- Dacă  $x_i \neq y_j$  și ultima transformare din  $s$  este o operație de înlocuire, atunci  $s_1 \dots s_{p-1}$  este o secvență optimă de transformări pentru  $X_{i-1}$  și  $Y_{j-1}$ .

Cele patru observații de mai sus ne indică substructura optimă a problemei noastre. Determinarea secvenței optimale de transformări pentru cele două șiruri  $X_i$  și  $Y_j$  se reduce la determinarea secvenței pentru una dintre perechile  $(X_{i-1}, Y_{j-1})$ ,  $(X_{i-1}, Y_j)$  sau  $(X_i, Y_{j-1})$ .

Ca de obicei, vom calcula mai întâi care este numărul de operații pentru a-l transforma pe  $x$  în  $y$ , urmând ca pe baza tabelului obținut să determinăm efectiv care este secvența optimă de transformări. Să notăm cu  $d_{ij}$  distanța Levenstein dintre  $X_i$  și  $Y_j$ . Pentru  $i = 0$ , vom avea nevoie de  $j$  operații de inserare pentru a ajunge la  $Y_j$  ( $X_0$  este șirul vid), deci  $d_{0j} = j$ ; similar avem  $d_{i0} = i$ .

Să găsim acum relația de recurență pentru cazul general. Conform primei observații, dacă  $x_i = y_j$ , vom avea  $d_{ij} = d_{i-1,j-1}$ . Dacă  $x_i \neq y_j$  atunci  $d_{ij}$  va fi, funcție de care dintre cele trei operații s-a aplicat, egal cu  $d_{i-1,j-1} + 1$ ,  $d_{i-1,j} + 1$ , sau  $d_{i,j-1} + 1$ . Vom alege desigur cea mai mică valoare. Astfel, formula de recurență pentru  $d_{ij}$  este:

$$d_{ij} = \begin{cases} j & \text{pentru } i = 0 \\ i & \text{pentru } j = 0 \\ d_{i-1,j-1} & \text{pentru } x_i = y_j \\ 1 + \min\{d_{i-1,j}, d_{i,j-1}, d_{i-1,j-1}\} & \text{pentru } x_i \neq y_j \end{cases}$$

Listing 14.18: Rezolvarea recurenței pentru distanța Levensthein

```

1 /**
2  * Calculeaza formula de recurenta pentru matricea d.
3  * @return distanta Levensthein dintre x si y
4  * care se presupun a fi membri ai clasei care contine metoda.
5  */
6 public static int distantaLevensthein()
7 {
8     int m = x.length();
9     int n = y.length();
10    d = new int[m + 1][n + 1];
11
12    for (int i = 0; i <= m; ++i)
13    {
14        d[i][0] = i;
15    }
16
17    for (int j = 0; j <= n; ++j)
18    {
19        d[0][j] = 0;
20    }
21
22    for (int i = 0; i < m; ++i)
23    {
24        for (int j = 0; j < n; ++j)
25        {
26            if (x.charAt(i) == y.charAt(j))
27            {
28                d[i + 1][j + 1] = d[i][j];
29            }
30            else
31            {
32                d[i + 1][j + 1] = 1 + min(d[i][j + 1], d[i + 1][j], d[i][j]);
33            }
34        }
35    }
36
37    return d[m][n];
38 }

```

Rezolvarea recurenței de mai sus este foarte simplă, și este realizată de metoda `distantaLevensthein()` din **Listing 14.18**.

Pentru a reconstitui secvența efectivă de operații prin care  $x$  se transformă în  $y$  vom folosi o parcurgere a matricei  $d$  asemănătoare cu cea de la problema subșirului comun de lungime maximă. Pornim de pe poziția  $(m, n)$ . Dacă  $x_m = y_n$ , atunci trecem pe poziția  $(m - 1, n - 1)$  fără nici o operație. Dacă

Figura 14.6: Elementele matricei  $d$  pentru șirurile *algorithm* și *aborigen*. Elementele îngroșate reprezintă traseul corespunzător transformării optime dată la începutul paragrafului.

	$\lambda$	a	b	o	r	i	g	e	n
$\lambda$	<b>0</b>	1	2	3	4	5	6	7	8
a	1	<b>0</b>	1	2	3	4	5	6	7
l	2	1	<b>1</b>	2	3	4	5	6	7
g	3	2	<b>2</b>	2	3	4	4	5	6
o	4	3	3	<b>2</b>	3	4	5	6	7
r	5	4	4	3	<b>2</b>	3	4	5	6
i	6	5	5	4	3	<b>2</b>	3	4	5
t	7	6	6	5	4	3	<b>3</b>	<b>4</b>	5
m	8	7	7	6	5	4	4	4	<b>5</b>

$x_m \neq y_n$ , atunci alegem poziția învecinată de valoare minimă. Dacă valoarea minimă s-a obținut pentru  $(m-1, n-1)$ , atunci ultima operație a fost de înlocuire; dacă valoarea minimă s-a obținut pentru elementul de pe poziția  $(m-1, n)$ , atunci ultima operație a fost de inserare, iar dacă valoarea minimă s-a obținut pentru  $(m, n-1)$ , ultima operație a fost de ștergere.

**Figura 14.6** prezintă matricea  $d$  pentru șirurile *algorithm* și *aborigen*, iar elementele îngroșate corespund soluției date la începutul paragrafului. Algoritmul de reconstituire este asemănător cu cel de la problema subșirului comun de lungime maximă din paragraful precedent și este redat în liniile 61-98 din **Listing 14.19**.

Listing 14.19: Soluția completă a problemei distanței Levensthein

```

1 import java.io.*;
2 import io.Reader;
3
4 public class Levensthein
5 {
6     private static String x;
7     private static String y;
8     private static int[][] d;
9
10    public static int distantaLevensthein()
11    {
12        int m = x.length();
13        int n = y.length();
14        d = new int[m + 1][n + 1];
15    }

```

```
16     for (int i = 0; i <= m; i++)
17     {
18         d[i][0] = i;
19     }
20
21     for (int j = 0; j <= n; j++)
22     {
23         d[0][j] = j;
24     }
25
26     for (int i = 0; i < m; i++)
27     {
28         for (int j = 0; j < n; j++)
29         {
30             if (x.charAt(i) == y.charAt(j))
31             {
32                 d[i+1][j+1] = d[i][j];
33             }
34             else
35             {
36                 d[i+1][j+1] = 1 + min(d[i][j+1],
37                                     d[i+1][j],
38                                     d[i][j]);
39             }
40         }
41     }
42
43     System.out.println("Operatiile de transformare: ");
44     determinaSecventa(m, n);
45
46     return d[m][n];
47 }
48
49 private static int min(int a, int b, int c)
50 {
51     int m = Math.min(a, b);
52
53     if (c < m)
54     {
55         m = c;
56     }
57
58     return m;
59 }
60
61 public static void determinaSecventa(int i, int j)
62 {
63     if (i != 0 && j != 0)
64     {
65         if (x.charAt(i-1) == y.charAt(j-1))
```

```

66     {
67         determinaSecventa(i-1, j-1);
68     }
69     else
70     {
71         int m = min(d[i-1][j-1],
72                     d[i-1][j],
73                     d[i][j-1]);
74
75         if (m == d[i-1][j-1])
76         {
77             System.out.println("- inlocuire: " +
78                                 x.charAt(i-1) + " cu " +
79                                 y.charAt(j-1));
80             determinaSecventa(i-1, j-1);
81         }
82         else if (m == d[i-1][j])
83         {
84             System.out.println("- stergere: " +
85                                 x.charAt(i-1) +
86                                 " de pe pozitia " + (i-1));
87             determinaSecventa(i-1, j);
88         }
89         else
90         {
91             System.out.println("- inserare: " +
92                                 y.charAt(j-1) +
93                                 " pe pozitia " + i);
94             determinaSecventa(i, j-1);
95         }
96     }
97 }
98 }
99
100 public static void main(String args[])
101 {
102     System.out.print("x = ");
103     x = Reader.readString();
104
105     System.out.print("y = ");
106     y = Reader.readString();
107     System.out.println("Numarul minim de operatii: "
108                       + distantaLevensthein());
109 }
110 }

```

## Rezumat

### Noțiuni fundamentale

**programare dinamică:** metodă de elaborare a algoritmilor care se aplică problemelor de optimizare în care soluția optimă se construiește pe baza soluției optime a subproblemelor.

**problemă de optimizare:** problemă în care se cere minimizarea sau maximizarea unei funcții obiectiv (de exemplu, în cazul înmulțirii șirului de matrice, funcția obiectiv este numărul de înmulțiri scalare necesare pentru a calcula produsul).

**principiul optimalității:** este numit adeseori și *substructură optimă*. În esență, acest principiu afirmă că soluția optimă a unei probleme de optimizare este construită din subsoluții optime ale subproblemelor.

**soluție optimă:** o soluție pentru care se atinge valoarea optimă a funcției obiectiv în cazul unei probleme de optimizare. Soluția optimă nu este neapărat unică.

**subproblemă:** problemă similară cu cea originală, dar de dimensiune mai mică. În cazul programării dinamice subproblemele se aleg astfel încât să respecte principiul optimalității.

### Erori frecvente

1. Se confundă principiul optimalității cu reciproca lui, crezând că prin combinarea a două subsoluții optime se obține tot o soluție optimă.
2. Se aplică metoda programării dinamice pentru subprobleme care nu respectă principiul optimalității.
3. Nu trebuie să ne avântăm să rezolvăm toate problemele care respectă principiul optimalității folosind programarea dinamică. Trebuie verificat înainte că nu putem aplica strategii mai simple, cum ar fi Greedy.
4. Suprapunerea apelurilor recursive trebuie evitată, deoarece există posibilitatea de a genera algoritmi exponențiali.
5. Calculul complexității în timp a algoritmilor recursivi se face pe baza unei formule recurente. Nu vă bazați pe faptul că un apel recursiv are timp liniar.

## Exerciții

### Pe scurt

1. Cărui tip de probleme se poate aplica metoda programării dinamice?
2. Care este enunțul principiului optimalității?
3. Este reciproca principiului optimalității adevărată?
4. Care este principala dificultate în rezolvarea problemelor de programare dinamică?
5. Construiți matricea 1 și evidențiați drumul de lungime maximă pentru triunghiul:
 

4				
2	9			
11	2	1		
4	8	9	6	
7	10	1	12	4
6. Găsiți un exemplu pentru care problema triunghiului din paragraful 14.3 admite mai multe soluții optime. Construiți matricea 1 atât pentru metoda înainte cât și pentru metoda înapoi.
7. Găsiți o parantezare optimă a produsului unui șir de matrice de dimensiuni  $(4, 12, 5, 9, 2, 60, 8)$ .
8. Determinați cel mai lung subșir comun pentru  $(1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 0)$  și  $(0, 0, 1, 1, 0, 1, 1, 0, 1, 0, 0, 1, 0, 1)$ .
9. Determinați distanța Levensthein între șirurile *recursivitate* și *acuitate*.

### Teorie

1. Demonstrați că o parantezare corectă a unui șir de  $n$  matrice are exact  $n - 1$  perechi de paranteze.
2. Care este modalitatea cea mai eficientă de determinare a parantezării optime a unui șir de matrice: enumerarea tuturor parantezărilor posibile ale produsului și calculul numărului de operații pentru fiecare parantezare, sau rezolvarea recursivă a formulei de recurență 14.1? Justificați răspunsul.

## Probleme

1. Toate problemele de programare dinamică prezentate în cadrul acestui capitol (problema triunghiului, parantezarea unui șir de matrice, subșir comun de lungime maximă etc.) admit în unele situații mai multe soluții optime. Totuși, algoritmi prezentați de noi determină o *singură* soluție optimă pentru fiecare problemă. Modificați metodele de determinare a soluției optime astfel încât acestea să afișeze *toate* soluțiile optime ale problemei.

### Indicație

Toți algoritmi de determinare a soluției optime se bazează pe parcurgerea ulterioară a matricei soluție reconstruind traseul pe care soluția optimă a fost obținută. De exemplu, în cazul problemei triunghiului, se pornește din vârful matricei 1 și se merge în jos sau dreapta-jos, funcție de care element este mai mare. Totuși, dacă cele două elemente sunt egale, înseamnă că soluția optimă se poate construi mergând pe *ambele* variante. Rezolvarea corectă este așadar să se parcurgă recursiv ambele variante și să se afișeze soluțiile corespunzătoare fiecăreia. Este important de observat că situația în care elementele sunt egale poate să apară de mai multe ori în cadrul reconstruirii soluției. Practic, de câte ori apare această situație, atâtea soluții optime admite problema.

### 2. Problema bancomatului.

Se pune problema construirii unui bancomat care să fie capabil să furnizeze o anumită sumă clienților băncii. Pentru aceasta, aparatul dispune de un număr  $k$  de bancnote. Deși fiecare dintre aceste bancnote există în cantitate nelimitată, se dorește returnarea restului cu un număr minim de bancnote. Se cere ca dându-se o sumă  $S$  și  $k$  numere reprezentând valorile bancnotelor ( $v_1, v_2, \dots, v_n$ ), să se determine numărul minim de bancnote cu care se poate plăti suma  $S$ , în cazul în care acest lucru este posibil. Să se determine și bancnotele cu care suma este plătită.

### Observație

Problema nu este trivială, întrucât bancnotele pot avea orice valoare naturală. De exemplu, nu se poate plăti suma 25 cu bancnote având valoarea 7 și 13). De asemenea, strategia Greedy de a folosi monede de valoare maximă nu dă întotdeauna soluția optimă (de exemplu, achitarea sumei 20 cu monede de valoare 6, 5, 4, 1 ar genera soluția (6, 6, 6, 1, 1), care nu este optimă).



**Indicație**

Notăm cu  $b_i$  numărul minim de bancnote cu care poate fi plătită suma  $i$ .  $b_s$  ne va da chiar soluția problemei (numărul minim de bancnote necesare pentru a plăti suma  $S$ ). Prin convenție,  $b_i$  va fi infinit dacă suma  $i$  nu poate fi plătită cu bancnotele date. Se inițializează elementele vectorului  $b$  cu valoarea infinit (`Integer.MAXINT` în cazul nostru). Pentru fiecare bancnotă cu valoarea  $v_k$  se pune  $b_{v_k} = 1$ . Apoi vom determina  $b_i$ , cu formula de recurență

$$b_i = \min_{i-v_k > 0} \{b_{i-v_k}\} + 1 \quad i = 1, 2, \dots, s$$

care exprimă faptul că suma  $i$  se poate obține adăugând moneda  $v_k$  la monedele necesare pentru a obține suma  $i - v_k$ . Reconstituirea soluției se face reținând pentru fiecare  $i$  indicele  $k$  pentru care s-a obținut valoarea minimă.

**3. Company party.**

Președintele unei companii dorește să organizeze o petrecere cu angajații firmei. Aceștia se află într-o structură ierarhică de subordonare care este reprezentată sub forma unui arbore a cărui rădăcină este, desigur, chiar președintele. Pentru ca toată lumea să se simtă bine, nu trebuie să fie invitate două persoane care să se afle în relație directă de șef-subaltern. Mai mult decât atât, fiecare angajat are un atribut calculat de serviciul personal - *rata de sociabilitate*. Se dorește ca suma ratelor de sociabilitate ale persoanelor invitate să fie maximă, asigurând astfel o petrecere mai mult decât reușită. Ca o condiție suplimentară, președintele trebuie să meargă la propria-i petrecere...

**Indicație**

Se consideră fiecare subarbore al ierarhiei date, calculând valoarea optimă în cazul în care vârful subarborelui este invitat și în cazul în care nu este invitat. Dacă vârful arborelui este invitat la petrecere, atunci subalternii săi direcți nu vor fi invitați. Dacă vârful nu este invitat, atunci nu există nici o restricție pentru subalternii direcți (care pot sau nu să fie invitați).

**4. Problema paragrafării.**

Se dau  $n$  cuvinte de lungime (număr de caractere)  $l_1, l_2, \dots, l_n$ . Trebuie să se așeze aceste cuvinte în ordine într-un paragraf în care lungimea

unei linii este  $l$ . Fiecare linie din paragraf trebuie să înceapă cu un cuvânt. Distanța optimă (numărul de spații) dintre două cuvinte este dată de un număr  $d$ . Distanța minimă dintre două cuvinte este dată de numărul  $d_{min}$ . Toate cele  $n$  cuvinte au lungime mai mică decât  $l$ . Pentru fiecare linie se calculează un cost care se obține prin însumarea abaterilor de la distanța optimă dintre cuvinte la care se adună numărul de spații libere rămase după ultimul cuvânt până la finalul liniei. Costul ultimei linii este 0 chiar dacă aceasta este incompletă. Se definește costul paragrafării ca fiind suma costurilor liniilor care formează textul. Se cere să se furnizeze o așezare a celor  $n$  cuvinte care să fie de cost minim.

### Exemplu

Se dau 4 cuvinte de lungime respectiv 1,2,2,6, lungimea liniei fiind 8. Se consideră distanța optimă  $d = 2$ , iar distanța minimă  $d_{min} = 1$ . Paragrafarea optimă este ( $c$  reprezintă un caracter din cuvânt, iar  $s$  reprezintă un spațiu):

*cs sccscc*

*cccccs s*

având costul 1.

### Indicație

Vom considera ca subproblemă de ordin  $i$  aranjarea în paragraf a cuvintelor  $i, i+1, \dots, n$ . Se observă ușor că dacă o soluție optimă conține o linie al cărei început este cuvântul  $i$ , atunci subsoluția este optimă pentru subproblema de ordin  $i$ . Vom nota cu  $c_i$  costul paragrafării cuvintelor  $i, i+1, \dots, n$ . Evident,  $c_n = 0$ , deoarece costul ultimei linii este 0. Formula de recurență pentru șirul  $c$  este

$$c_i = \begin{cases} 0, & \text{pentru } i = n \\ \min_{j=i, n-1} \{a_{ij} + c_{j+1}\}, & \text{altfel} \end{cases}$$

unde  $a_{ij}$  este costul obținut prin așezarea cuvintelor  $i, i+1, \dots, j$  pe o linie (în situația în care cuvintele nu pot fi așezate pe linie, costul se consideră a fi infinit).  $a_{ij}$  se află ușor calculând diferența dintre spațiile disponibile ( $l - \sum_{k=i}^j l_k$ ) și cele care ar trebui să fie pentru un cost 0  $((j-i) \cdot d_{opt})$ :

$$a_{ij} = \begin{cases} \infty, & \text{pentru } \sum_{k=i}^j l_k + (j-i) \cdot d_{min} > l \\ \left| l - \sum_{k=i}^j l_k - (j-i) \cdot d_{opt} \right|, & \text{altfel} \end{cases}.$$

Aflarea unei paragrafări optime se face reținând cuvintele la care se trece pe linie nouă și distribuind uniform spațiile avute la dispoziție în cadrul cuvintelor din aceeași linie.

### 5. Mere, pere.

Se consideră  $n$  camere distincte, situate succesiv una după cealaltă astfel încât din camera numărul  $i$  se poate trece doar în camera numărul  $i + 1$  ( $i = 1, 2, \dots, n-1$ ). În fiecare cameră se află un anumit număr de mere și de pere. O persoană având la dispoziție un rucsac suficient de încăpător, inițial gol, pornește din camera 1, trece prin camerele 2, 3,  $\dots$ ,  $n$  și iese. La intrarea în fiecare cameră persoana trebuie să descarce rucsacul și să încarce fie toate merele, fie toate perele din camera respectivă, după care trece în următoarea cameră. Se presupune că pentru fiecare fruct transportat dintr-o cameră într-alta persoana consumă câte o calorie. Să se precizeze ce fructe trebuie să încarce persoana respectivă în fiecare cameră astfel încât după parcurgerea celor  $n$  camere să consume un număr minim de calorii și să se precizeze acest număr.

#### Indicație

Strategia Greedy de a alege la fiecare pas cantitatea de fructe mai mică nu conduce întotdeauna la soluția optimă (găsiți un contraexemplu!). Principiul optimalității se formulează astfel: presupunem că avem o soluție optimă pentru subproblema generată de camerele  $i, i + 1, \dots, n$ ; atunci subsoluția acesteia pornind din camera  $i + 1$  este optimă pentru subproblema generată de camerele  $i + 1, i + 2, \dots, n$  **în care se pleacă cu fructele alese în camera  $i + 1$  în cadrul soluției optime**. Cu alte cuvinte, dacă în camera  $i + 1$  s-au ales mere, atunci subsoluția respectivă este optimă pentru subproblema generată de camerele  $i + 1, i + 2, \dots, n$  în care se presupune că se pornește cu mere (dacă se pornește cu pere se poate obține o soluție mai bună).

Vom nota cu  $m_i$  numărul merelor și cu  $p_i$  numărul perelor din camera  $i$ ,  $i = 1, 2, \dots, n$ . De asemenea, vom nota cu  $cm_i$  costul optim (numărul minim de calorii) obținut când se pleacă cu mere din camera  $i$  și cu  $cp_i$  costul optim obținut când se pleacă cu pere din camera  $i$ . Evident, soluția problemei este dată de  $\min(cp_1, cm_1)$ . Formula de recurență este dată de observația că dacă plecăm cu un anumit tip de fruct din camera  $i$ , putem

să schimbăm acel tip de fruct în oricare dintre camerele  $i+1, i+2, \dots, n$ :

$$cm_i = \left\{ \min \left\{ \begin{array}{l} m_n, \quad i = n \\ m_i + cp_{i+1} \\ m_i + (m_i + m_{i+1}) + cp_{i+2} \\ \dots\dots\dots \\ m_i + (m_i + m_{i+1}) + \dots + (m_i + \dots m_{n-1}) + cp_n \\ m_i + (m_i + m_{i+1}) + \dots + (m_i + \dots m_n) \end{array} \right. \right.$$

Analog se obține formula de recurență și pentru pere.

#### 6. Mere, pere - problema generalizată.

Generalizați problema anterioară pentru cazul în care în fiecare cameră se găsesc  $t$  tipuri de fructe.

#### 7. Compararea secvențelor de ADN.

Se numește secvență de ADN un șir format din caracterele A, C, T și G (corespunzătoare elementelor fundamentale ale codului genetic: adenină, citozină, timină și guanină). O problemă de o deosebită importanță în genetică este compararea a două secvențe de ADN (de exemplu, pentru stabilirea paternității). Compararea a două secvențe de ADN nu se face însă direct, existând posibilitatea alinierii celor două șiruri prin inserarea de caractere '-' (după aliniere cele două șiruri au aceeași dimensiune). De exemplu, dacă avem secvențele TGCGAT și TAGCAG, o posibilă aliniere este:

*T - GCGAT*

*TAGC - AG*

Se acordă 3 puncte dacă două caractere care se află unul sub celălalt sunt egale și se acordă o penalizare de un punct dacă cele două caractere sunt diferite. Pentru exemplul de mai sus se acordă  $4 \cdot 3 - 3 \cdot 1 = 9$  puncte.

Fiind date două șiruri ADN, A și B (nefiind obligatoriu ca acestea să aibă aceeași dimensiune), se cere să se determine o aliniere pentru care punctajul acordat este maxim. (Pentru exemplul precedent, alinierea dată este optimă).

**Indicație**

Această problemă seamănă întrucâtva cu problema determinării subșirului comun de lungime maximă (paragraful 14.6). O primă idee de rezolvare ar fi să determinăm subșirul comun de lungime maximă pentru cele două șiruri și apoi să potrivim elementele comune din cele două șiruri unele sub altele, folosindu-ne de '-' atât sus cât și jos. Această soluție ar fi corectă dacă nu ar exista penalizări pentru nepotriviri. Cu toate acestea, structura de optimalitate este asemănătoare cu cea a problemei subșirului comun de lungime maximă, diferind doar relația de recurență. Și aici vom considera prefixe ale celor două șiruri, iar pentru fiecare pereche de prefixe vom determina alinierea optimală. Prin aliniere se înțelege o reprezentare a ieșirii cerute de problemă în care se inserează caractere '-' pentru ca șirurile să aibă aceeași lungime.

Fie  $s_{kl}$  scorul maxim ce se poate obține prin alinierea prefixelor  $A_1 \dots A_k$  și  $B_1 \dots B_l$ . Dacă  $A_k = B_l$ , atunci  $s_{kl}$  va fi  $s_{k-1, l-1} + 3$ , deoarece se acordă 3 puncte pentru două caractere identice așezate unul sub altul. Dacă cele două caractere nu sunt egale, atunci putem obține  $s_{kl}$  fie din  $s_{k-1, l}$ , fie din  $s_{k, l-1}$  sau din  $s_{k-1, l-1}$  corespunzător respectiv situației în care adăugăm un '-' la șirul A, un '-' la șirul B sau câte un caracter diferit de '-' la ambele șiruri. Pentru fiecare dintre aceste trei cazuri va trebui acordată o penalizare pentru nepotrivirea caracterelor. Desigur, vom alege varianta al cărei cost este maxim.

Formula de recurență pentru matricea  $s$

$$s_{kl} = \begin{cases} -k, & \text{pentru } l = 0 \\ -l, & \text{pentru } k = 0 \\ s_{k-1, l-1} + 3, & \text{pentru } A_k = B_l \\ \max\{s_{k-1, l}, s_{k, l-1}, s_{k-1, l-1}\} - 1, & \text{pentru } A_k \neq B_l \end{cases}$$

Reconstituirea soluției se face ușor parcurgând matricea  $s$  de la  $s_{mn}$  până la  $s_{00}$ . Dacă  $s_{kl}$  provine din  $s_{k-1, l-1}$ , atunci este vorba de o așezare unul sub altul a două caractere ADN (deci nu '-'), momentan nefiind important dacă  $A_k$  este egal cu  $B_l$  sau nu. Dacă  $s_{kl}$  provine din  $s_{k-1, l}$ , atunci trebuie inserat un '-' în șirul B, altfel el trebuie inserat în A.

**8. Problema vrăjitorului.**

Indiana Jones intră într-un labirint unde găsește un vrăjitor. Acesta îi pune în față  $n$  lăzi, fiecare ladă conținând un anumit număr precizat ( $m_i$ )

de monede de aur. Vrajitorul îi cere să aleagă niște lăzi astfel încât suma monedelor din acestea să fie divizibilă cu  $n$ , altfel nu va putea pleca cu ele. Evident, Indiana Jones dorește să ia un număr cât mai mare de monede de aur. Elaborați un algoritm care să-l ajute pe Indiana Jones să plece cu cantitatea maximă de monede, respectând cerințele vrajitorului.

### Indicație

Trebuie să arătăm în primul rând că problema noastră admite soluție, cu alte cuvinte există cel puțin o combinație de lăzi pentru care suma monezilor este divizibilă cu  $n$ . Notăm cu  $s_i$  suma monezilor din lăzile  $1, 2, \dots, i$

$$s_i = \sum_{j=1}^i m_j$$

și cu  $r_i$  restul împărțirii lui  $s_i$  la  $n$ . Evident,  $r_i$  va lua valori întregi între 0 și  $n - 1$ . Dacă există un  $i$  pentru care avem  $r_i = 0$ , atunci rezultă că  $s_i$  se divide cu  $n$ , deci lăzile  $1, 2, \dots, i$  reprezintă o soluție a problemei. Dacă nu există nici un  $i$  pentru care  $r_i = 0$ , atunci rezultă că  $r_i$  ia valori între 1 și  $n - 1$ , deci vor exista cel puțin două componente  $r_i$  și  $r_j$  din șirul  $r$  (care are  $n$  componente) care să fie egale. Aceasta înseamnă că  $s_j - s_i$  se divide cu  $n$ , deci lăzile  $i + 1, \dots, j$  reprezintă o soluție a problemei. Am arătat deci că problema are întotdeauna măcar o soluție. Desigur, această metodă de alegere a lăzilor nu garantează optimalitatea soluției, motiv pentru care vom folosi metoda programării dinamice pentru a rezolva problema.

Vom nota cu  $M_{ij}$  suma maximă de monezi care se poate obține din primele  $i$  lăzi și care să dea restul  $j$  la împărțirea cu  $n$ . Prin convenție,  $M_{ij}$  va fi infinit dacă nu se poate obține o sumă care să dea restul  $j$  la împărțirea cu  $n$ . Scopul nostru este de a-l calcula pe  $M_{n0}$ . Este ușor de observat că  $M_{1j}$  este egal cu  $m_1$  dacă  $m_1$  se divide cu  $j$ , și infinit în caz contrar. Totuși, în cazul în care  $m_1$  nu se divide cu  $n$ , vom considera  $M_{10}$  ca fiind 0. În cazul general, pentru a calcula  $M_{kl}$  ne vom folosi de informația determinată pentru primele  $k - 1$  lăzi. Lada numărul  $k$  poate sau nu să apară în soluția subproblemei  $M_{kl}$ . Dacă nu este luată, atunci vom avea  $M_{kl} = M_{k-1l}$ . Dacă lada numărul  $k$  este luată, atunci  $M_{kl}$  va fi egal cu

$$M_{jp} + m_k$$

unde  $j$  este mai mic decât  $k$ , iar  $p + m_k$  trebuie să dea restul  $l$  la împărțirea la  $n$ . Formula de mai sus rezultă din observația că  $M_{jp}$  reprezintă suma

maximă care se poate obține din primele  $j$  lăzi care să fie divizibilă cu  $p$ . Dacă acestei sume  $i$  se adaugă  $m_k$ , se obține o sumă care prin împărțirea la  $n$  va da restul  $l$ , adică chiar  $M_{kl}$  (la acest pas se aplică principiul optimalității). Dintre toate aceste variante o vom alege, desigur, pe cea mai convenabilă

$$M_{kl} = \max\{M_{k-1l}, \max_{j=1}^{k-1}\{M_{jp} + m_k | j < k \text{ și } l = p + m_k \pmod{n}\}\}$$

Valoarea lui  $p$  din formula de mai sus poate fi calculată cu ușurință după formula  $p = (l - m_k) \pmod{n}$  (atenție la cazul în care  $l - m_k$  este negativ - puteți adăuga pentru siguranță un  $n$  la sumă). Reconstituirea soluției se face reținând pentru fiecare element  $M_{kl}$  o valoare  $p_{kl}$  care indică lada  $j$  pentru care s-a obținut valoarea optimă, sau este 0 dacă  $M_{kl} = M_{k-1l}$ .  
cu

## 15. Metoda Branch & bound

Trebuie să fii foarte atent dacă nu știi unde mergi, pentru că riști să nu ajungi acolo.

---

Yogi Berra

Pe parcursul acestui capitol vom vedea:

- În ce constă metoda Branch & bound de elaborare a algoritmilor;
- Care este legătura dintre Branch & bound și Backtracking;
- Abordarea unei probleme clasice de Branch & bound, cunoscută sub numele de jocul de puzzle cu 15 elemente.

### 15.1 Prezentare generală

Metoda Branch & bound este asemănătoare metodei Backtracking, în sensul că încearcă să facă o căutare exhaustivă inteligentă în spațiul soluțiilor (numit în acest context și *spațiul stărilor*) problemei. Comparativ cu Branch & bound, Backtracking acționează “orbește”, în sensul că la fiecare pas se alege (la întâmplare) o componentă în vectorul soluție care respectă condițiile de continuare. Singura condiție la Backtracking este așadar respectarea condițiilor de continuare: toate posibilele configurații care respectă această condiție sunt privite în mod egal și se alege arbitrar una dintre ele. Alegerea în cazul Branch & bound este ceva mai rafinată: fiecărei configurații viabile i se acordă o pondere care reprezintă o estimare a valorii acelei configurații. La fiecare pas vom alege configurația a cărei estimare este optimă. Ca și în cazul metodei Backtracking, problemele rezolvabile prin această metodă generează un spațiu al stărilor,



în care se găsesc toate configurațiile viabile posibile. Spațiul stărilor poate fi reprezentat în acest caz sub forma unui arbore, în care rădăcina arborelui este configurația inițială a problemei, în timp ce configurațiile finale constituie, în cazul în care există, frunzele arborelui. Este posibil ca pentru o anumită problemă să nu existe o soluție, adică să nu existe o secvență de mutări posibile, prin care să se ajungă de la o configurație inițială la o configurație dorită (finală). Asemănarea cu Backtracking este și acum evidentă. În cazul metodei Backtracking spațiul stărilor era reprezentat sub forma unei *liste* care începea cu configurația inițială și se încheia cu configurația finală în care nu se mai putea aplica nici una dintre cele patru transformări (atribuie și avansează, încercare eșuată, revenire, revenire după găsirea unei soluții). Diferența esențială dintre Backtracking și Branch&bound apare în cadrul pasului *atribuie și avansează*. În cazul Backtracking alegeam o configurație arbitrară care respecta condițiile de continuare. În cazul Branch & bound considerăm *toate* configurațiile pe care putem avansa și o alegem pe cea care pare să fie mai promițătoare.

Branch & bound găsește soluția optimală prin alegerea celei mai bune soluții de moment. Dacă soluția parțială curentă nu este mai potrivită decât cea mai bună soluție parțială disponibilă la un moment dat, atunci ea este abandonată. De exemplu, să presupunem că se dorește aflarea celui mai scurt drum de la Brașov la Constanța și că cea mai scurtă cale descoperită până la un moment dat este de 400 de kilometri. Apoi, să presupunem că vrem să considerăm posibilele rute care trec prin Galați. Dacă cel mai scurt drum din Brașov la Galați are 380 de kilometri, iar distanța dintre Galați și Constanța este de 50 de kilometri, atunci nu are nici un sens să căutăm drumuri către Constanța care trec prin Galați, pentru că ele vor fi tot timpul mai lungi decât cea mai scurtă cale cunoscută până în acel moment ( $380 + 50 > 400$ ). De aceea, rutele din Brașov către Constanța prin Galați nu vor mai fi explorate. Cu alte cuvinte, se încearcă parcurgerea subarborilor din spațiul soluțiilor care pot conduce la un rezultat favorabil, evitându-se subarborii despre care se află că nu conduc la rezultate optime.

### 15.1.1 Fundamente teoretice

Metoda Branch & bound utilizează câteva noțiuni a căror înțelegere este strict necesară pentru a putea deprinde mecanismul de funcționare a metodei.

Unei probleme de tipul Branch & bound i se asociază un *arbore oarecare* (nu neapărat arbore binar), în care fiecare nod reprezintă o *configurație*. Configurația inițială reprezintă rădăcina arborelui și conține datele de intrare ale problemei. Prin efectuarea operațiilor (mutărilor) permise asupra configurației inițiale se obțin alte configurații, care vor constitui nodurile aflate pe nivelul 2 al

arborelui. Procedul se repetă, obținându-se noi și noi configurații, care conduc la crearea unei structuri arborescente, care va conține soluția problemei (dacă aceasta există).

*Succesorul* unui nod reprezintă o configurație la care se ajunge prin aplicarea nodului respectiv a uneia dintre mutările permise de problemă. În cele mai multe probleme, un nod are mai mulți succesori.

*Expandarea* unui nod constituie acțiunea de determinare a tuturor succesorilor nodului respectiv. Nodul care este expandat este un nod de tip “tată”, în timp ce nodurile rezultate în urma expandării sunt noduri de tip “fiu”.

*Nodul răspuns* reprezintă configurația la care se dorește să se ajungă (altfel spus, configurația finală). Acesta este obținut prin aplicarea unei succesiuni de mutări configurației inițiale. Există și situații în care nu există o succesiune de mutări prin care să se ajungă la configurația finală. În acest caz, spunem că problema nu are soluție.

*Nodul terminal (frunză)* este un nod care nu are succesori. Dacă la un moment dat nu se mai poate realiza nici o mutare pe configurația curentă sau s-a ajuns la configurația finală, atunci ea nu va mai avea nici un succesor.

Există două tipuri de noduri în arborele asociat problemei Branch & bound: noduri *active* și noduri *expandate*.

Nodurile *active* reprezintă mulțimea de noduri obținută prin expandarea unui nod. Aceste noduri sunt denumite și noduri *încă neexpandate*.

Un nod *expandat* este nodul pe care se realizează operația de expandare. Practic, pe configurația reprezentată de acest nod se aplică toate operațiile permise de problemă, obținându-se o mulțime de noduri active. Alegerea nodului expandat este o problemă dificilă, deși nodul expandat este ales doar dintre elementele mulțimii nodurilor active existente în acel moment. Primul nod expandat este reprezentat de configurația inițială. Următoarele noduri expandate sunt alese pe baza unui procedeu care va fi prezentat mai târziu. Nodul expandat este întâlnit în literatura de specialitate și sub numele de nod *E* (engl. *E-node*) sau nod *curent*.

Nodurile active sunt stocate într-o listă. Modul de reprezentare al listei nodurilor active ne oferă informații despre modul de parcurgere al arborelui soluțiilor. Dacă nodurile active sunt păstrate într-o stivă (structură LIFO, Last In First Out), atunci arborele spațiului de stări va fi parcurs în adâncime (engl. DFS = Depth First Search), iar dacă nodurile sunt păstrate într-o coadă (structură FIFO, First In First Out), atunci arborele spațiului de stări va fi parcurs în lățime (engl. BFS = Breadth First Search).

Nu există un criteriu, care odată aplicat, să permită găsirea următorului nod care trebuie expandat. Din acest motiv se apelează la un procedeu specific inteligenței artificiale, prin care se asociază fiecărui nod o funcție *c* (funcție

*de cost*). Ca și funcția de continuare din cazul metodei Backtracking, aceasta este o funcție de limitare, utilizată pentru evitarea generării unor subarbori care nu au cum să conducă la nodul răspuns. De aici și numele metodei: *branch* înseamnă ramură, despărțitură, în timp ce *bound* înseamnă margine, limită, graniță. Alegerea acestei funcții este partea cea mai dificilă a metodei, pe ea bazându-se și selectarea următorului nod expandat.

Pe scurt, un algoritm Branch & bound funcționează la modul următor: din lista nodurilor active se va alege unul care devine nod expandat. Acest nod este următorul nod care va fi prelucrat. Nu pot exista mai multe noduri expandate în același timp. În momentul în care un nod devine nod expandat, se vor genera (determina) toate nodurile descendente (noduri succesori), acestea devenind noduri active. Ele se vor adăuga listei nodurilor active existente. Odată generați descendenții nodului expandat, va fi ales acela care este situat cel mai aproape de configurația finală sau, mai corect spus, cel care are cele mai mari șanse de a se afla mai aproape de starea finală. Pentru a afla care este acest nod, se definește funcția de cost  $c$  pe mulțimea nodurilor din arborele spațiului soluțiilor. Această funcție diferă de la problemă la problemă și trebuie aleasă în funcție de cerințele problemei respective, dar descoperirea ei este un pas important spre rezolvarea problemei. Odată ce am determinat funcția de cost, vom alege dintre nodurile active pe cel cu costul minim (pentru care funcția ia valoarea minimă), care devine astfel nod expandat. Parcurgerea de acest tip poartă numele de *căutare LC* (căutare Least Cost = căutare după costul minim). Altfel spus, atunci când căutăm o soluție în arborele generat de spațiul configurațiilor folosind funcția de limitare, spunem că efectuăm o căutare LC a soluției problemei, indiferent dacă arborele în sine este parcurs în adâncime (DFS) sau în lățime (BFS). Căutarea LC nu *înlocuiește* căutările în lățime sau în adâncime, ci doar le *îmbunătățește*. Funcția de cost nu poate fi determinată în mod precis, deoarece determinarea ei exactă este echivalentă cu rezolvarea problemei inițiale. Chiar dacă nu putem determina o funcție de cost infailibilă, există câteva sugestii care pot fi urmate în definirea ei:

- să poată fi calculată pentru un nod doar pe baza informațiilor deținute de nodurile intermediare aflate pe drumul către rădăcina arborelui;
- să fie independentă de generarea nodurilor active ale nodului expandat.

Figura 15.1: Un posibil aranjament al jocului de puzzle cu 15 elemente.

1	2	3	4
5	6		8
9	10	7	11
13	14	15	12

Figura 15.2: Aranjamentul final al jocului de puzzle cu 15 elemente.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

## 15.2 Un exemplu: Puzzle cu 15 elemente

### 15.2.1 Enunțul problemei

Puzzle cu 15 elemente este un joc distractiv inventat de Sam Loyd în 1878, care constă într-un tabel cu 16 celule (căsuțe), dintre care 15 celule sunt numerotate de la 1 la 15, iar o celulă rămâne liberă (**Figura 15.1**).

Regulile acestui joc sunt simple. Inițial se dă un aranjament de tipul celui din **Figura 15.1**, iar scopul jocului este de a transforma acest aranjament prin transformări (mutări) succesive în aranjamentul din **Figura 15.2**.

Aranjamentul inițial se mai numește *configurație inițială* sau *stare inițială*, iar cel final *configurație finală* sau *stare finală*. Pentru a ajunge de la configurația inițială la cea finală este permis doar un singur tip de mutare: o celulă vecină cu celula liberă poate să facă schimb de poziții cu aceasta. Astfel, să considerăm aranjamentul din **Figura 15.1** ca fiind configurația inițială. Atunci există patru mutări permise de regulamentul jocului: se poate muta celula liberă în locul uneia din celulele care conțin valorile 3, 6, 8, 7, ca în **Figura 15.3**

După cum se poate observa, se consideră a fi celule vecine ale celulei libere, doar cele care au o latură comună cu celulă respectivă (în cazul configurației noastre inițiale este vorba de căsuțele cu valorile 3, 6, 7, 8). Căsuțele cu valorile 2, 4, 10, 12 *nu* sunt considerate celule vecine ale celulei libere. Evident, nu tot timpul sunt posibile toate cele patru mutări. În situația în care celula liberă nu

Figura 15.3: Mutările permise pentru celula liberă din tabela 15.1.

sus				dreapta			
1	2		4	1	2	3	4
5	6	3	8	5	6	8	
9	10	7	11	9	10	7	11
13	14	15	12	13	14	15	12
jos				stânga			
1	2	3	4	1	2	3	4
5	6	7	8	5		6	8
9	10		11	9	10	7	11
13	14	15	12	13	14	15	12

are patru celule vecine (de exemplu, dacă se află pe prima linie, sau pe prima coloană) mutările respective nu sunt posibile.

Asemănător, pe fiecare dintre cele patru configurații obținute se pot realiza alte mutări, generându-se astfel un nou set de configurații. Toate aceste configurații poartă numele de stări ale jocului. O stare poate fi obținută pornind de la starea inițială căreia i se aplică o secvență de mutări permise. Spațiul stărilor unei configurații inițiale reprezintă mulțimea tuturor stărilor care pot fi obținute pornind de la starea inițială.

Generarea configurațiilor continuă până când se ajunge la configurația finală. Uneori este posibil ca starea finală să nu poată fi obținută pornind de la starea inițială, caz în care problema noastră nu va avea soluție.

### 15.2.2 Rezolvarea problemei

Cea mai directă cale de a rezolva un astfel de puzzle este ca pornind de la starea inițială să se genereze toate configurațiile intermediare până când se obține configurația finală, sau s-a epuizat întreg spațiul de căutare. Este ușor de observat că numărul total al configurațiilor jocului este de  $16! \approx 20.9 \cdot 10^{12}$ . Prin urmare, spațiul stărilor pentru problema noastră este foarte mare și o căutare exhaustivă este sortită eșecului.

Înainte de a încerca să căutăm configurația finală în cadrul spațiului stărilor, este util să determinăm dacă starea finală *poate* fi obținută printr-o secvență de mutări din configurația inițială. Pentru aceasta vom nota căsuțele de la 1 la 16. Astfel, în configurația inițială din **Figura 15.1** celula 1 conține valoarea 1, celula 11 conține valoarea 7, celula 12 conține valoarea 11, etc. Pentru configurația

Figura 15.4: Valoarea variabilei  $x$ , funcție de poziția celulei libere în configurația inițială

	*		*
*		*	
	*		*
*		*	

finală, cele două valori sunt egale: celula 1 conține valoarea 1, celula 2 conține valoarea 2 etc... Celulei libere  $i$  se atribuie valoarea 16.

Considerăm funcția  $position(i)$  ca fiind numărul celulei care conține valoarea  $i$ . De exemplu, în configurația inițială prezentată în **Figura 15.1**, avem:

- $position(3) = 3$
- $position(7) = 11$
- $position(11) = 12$  etc.

Pentru celula liberă avem  $position(16) = 7$ , ceea ce înseamnă că spațiul liber se află pe poziția 7 în configurația inițială.

De asemenea, considerăm funcția  $less(i)$  ca fiind numărul de celule  $j$ , cu  $j < i$ , pentru care  $position(j) > position(i)$ . Cu alte cuvinte, această funcție calculează câte celule cu valori mai mici decât valoarea celulei curente se găsesc după ea în configurația inițială. De exemplu, în **Figura 15.1**, pentru celula care conține valoarea 8, există o celulă cu valoare mai mică aflată pe o poziție mai mare, și anume, celula cu valoarea 7. Așadar,  $less(8)=1$ . Analog, se pot determina și celelalte valori:  $less(13)=1$ ,  $less(5)=0$ ,  $less(11)=0$ , etc.

Ultima considerație se referă la poziția spațiului liber în cadrul configurației inițiale. Considerăm o variabilă  $x$  care are valoarea 1, dacă spațiul liber se află în configurația inițială pe una din pozițiile marcate cu \* în **Figura 15.4**, sau 0, dacă spațiul liber se află pe una dintre pozițiile nemarcate în aceeași figură.

Având aceste notații, putem enunța următorul rezultat:

**Teorema 15.2.1** Configurația finală poate fi obținută din configurația inițială dacă și numai dacă

$$x + \sum_{i=1}^{16} less(i) \text{ este un număr par.}$$

Demonstrația acestei teoreme se bazează pe observații simple asupra configurațiilor posibile și o puteți găsi pe multe situri care tratează această problemă

(de exemplu <http://kevingong.com/Math/SixteenPuzzle.html>). Putem folosi această teoremă pentru a determina dacă starea finală se află în spațiul stărilor asociat stării inițiale. Dacă starea finală poate fi obținută plecând de la starea inițială, atunci are sens să începem căutarea secvenței de mutări permise care ne conduce la starea finală.

Pentru a realiza această căutare este util să organizăm spațiul stărilor într-un arbore. Rădăcina arborelui este starea (configurația) inițială. Copiii fiecărui nod  $X$  al arborelui reprezintă stări care pot fi obținute din starea  $X$  printr-o mutare permisă de regulamentul jocului. Pentru simplitatea notației, este mai convenabil să ne gândim la o mutare ca fiind mai degrabă o mutare a celulei libere decât a unei alte celule. Prin urmare, celula liberă poate fi mutată la fiecare mutare, fie în sus, în jos, la stânga sau la dreapta, dacă este posibil.

Iată cum ar arăta arborele spațiului de stări după câteva mutări în configurația inițială:

După cum se poate observa și din **Figura 15.5**, prin mutări succesive s-au obținut diverse stări intermediare, care au creat o structură arborescentă: starea  $A$  a generat stările  $B$ ,  $C$ ,  $D$  și  $E$  prin mutări în sus, respectiv la dreapta, în jos și la stânga. Cele patru stări nou create, la rândul lor, au generat alte stări (de exemplu, starea  $D$  a generat stările  $J$  și  $K$  prin mutări la dreapta și în jos). În final din starea  $J$  s-a generat starea  $M$  prin mutarea spațiului liber în jos, atingându-se starea finală, situație în care generarea unor noi mutări este stopată. Se poate observa că au fost necesare trei mutări pentru a atinge starea finală pornind de la starea inițială. Figura anterioară nu este completă, în sensul că, pentru a păstra simplitatea, am ales să nu prezentăm toate mutările care ar fi fost posibile. De exemplu, configurația  $D$  generează de fapt patru alte stări (prin mutări în sus, în jos, la stânga și la dreapta ale spațiului liber), dintre care noi am prezentat doar două ( $J$  și  $K$ ).

Dacă am fi generat arborele stărilor folosind una dintre cele două posibilități, parcurgerea în lățime (în care se generează toate stările obținute din stările  $B$ ,  $C$ ,  $D$ ,  $E$ , apoi toate stările pentru  $F$ ,  $G$ ,  $H$  etc.) sau cea în adâncime (în care se generează toate stările obținute din starea  $B$ , apoi toate stările obținute din starea  $F$  etc.), ne-am fi îndepărtat simțitor de soluția problemei. Acest lucru este pus în evidență foarte clare de **Figura 15.5**. Asta se întâmplă pentru că, folosind parcurgerea spațiului de configurații în lățime sau în adâncime, căutarea soluției este “oarbă”, adică nu ține cont de configurația inițială și nici de cele intermediare obținute, pentru a vedea care dintre ele este mai aproape de configurația finală. Practic, procedeul ar genera aceeași secvență de stări indiferent de starea inițială a problemei. Căutarea în adâncime corespunde în mod precis parcurgerii spațiului de căutare rezultat în urma aplicării metodei *Backtracking*. Coborârea în arbore se asociază pasului *atribuie și avansează*, iar revenirea la nodul părinte

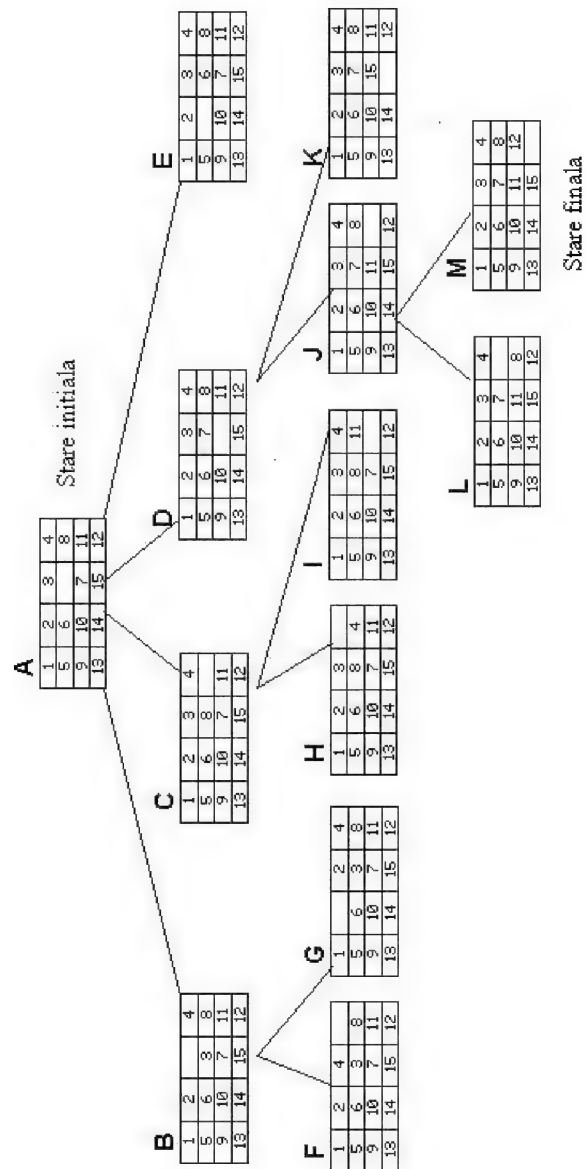


Figura 15.5: O parte a arborelui stărilor asociat stării inițiale din **Figura 15.1**. Se pot observa starea inițială și cea finală.



se asociază pasului de *revenire*.

Pentru a evita căutarea oarbă a soluției, am dori să utilizăm o metodă de căutare “inteligentă”. Această metodă inteligentă ar trebui să caute nodul răspuns (configurația finală) și să adapteze calea către acesta, în funcție de starea inițială de la care pornim căutarea. O metodă pentru evitarea căutării brute este să asociem o funcție de cost fiecărui nod din spațiul stărilor, funcție care să scoată cumva în evidență nodurile care sunt mai “promițătoare”, deci care au șanse mai mari să ne conducă la soluția problemei.

Așadar, fiecărui nod  $X$  din arborele stărilor îi asociem o funcție  $c$ , iar  $c(X)$  reprezintă costul nodului  $X$ . Funcția  $c$  o definim ca o sumă de alte două funcții  $f$  și  $g$ , unde  $f(X)$  o definim ca fiind lungimea drumului de la rădăcina arborelui de stări (altfel spus, de la configurația inițială) până la nodul  $X$ , iar  $g(X)$  este o estimare a lungimii căii celei mai scurte de la nodul  $X$  la un nod răspuns aflat în subarborele de rădăcină  $X$ . O posibilă alegere pentru  $g(X)$  este numărul de celule nelibere care nu se află pe poziția din configurația finală. Aceasta este o alegere naturală, din moment ce este clar că pentru a ajunge la configurația finală trebuie realizate cel puțin  $g(X)$  mutări pe starea  $X$ . Pentru o mai bună înțelegere a funcției de cost, iată câteva valori calculate pentru arborele de stări din **Figura 15.5**:  $c(B) = 1 + 4 = 5$  ( $f(B) = 1$  pentru că nodul  $B$  se află pe nivelul 2 în arbore, deci la o distanță de un (1) nivel față de rădăcină, iar  $g(B) = 4$  pentru că sunt 4 valori care nu se află în pozițiile lor finale: 3, 7, 11, 12),  $c(C) = 1 + 5 = 6$ ,  $c(D) = 1 + 2 = 3$  etc.

Folosind funcția  $c$  definită anterior, vom realiza o căutare LC a nodului răspuns pe baza algoritmului descris în paragraful 15.1.1. După cum s-a putut observa în cazul căutărilor nodului răspuns prin intermediul parcurgerii standard în lățime sau în adâncime a arborelui de stări, selecția nodului expandat nu s-a realizat ținând cont de șansele de a ajunge la nodul răspuns într-un timp cât mai redus. Căutarea nodului răspuns este “îmbunătățită” din punct de vedere al vitezei prin utilizarea unei funcții inteligente de clasificare a nodurilor, așa cum este cazul funcției de cost  $c$ . Următorul nod expandat va fi selectat, în algoritmul pe care îl propunem, nu la întâmplare (a se citi “indiferent de configurația reprezentată de nodul respectiv”), ci pe baza valorilor pe care posibilele noduri expandabile le vor avea prin aplicarea funcției de cost.

Căutarea LC pornește de la configurația inițială (**Figura 15.1**). Conform algoritmului, acest nod este primul nod expandat (nodul A din **Figura 15.5**). Se generează toți fiii nodului expandat, adică nodurile B, C, D și E. Aceste patru noduri reprezintă noduri active și vor fi adăugate în lista nodurilor active. Din lista nodurilor active existente, se va alege viitorul nod expandat. Pentru aceasta se calculează  $c(B)$ ,  $c(C)$ ,  $c(D)$ ,  $c(E)$ , alegându-se nodul pentru care funcția  $c$  are valoarea minimă. Cum  $c(B) = 5$ ,  $c(C) = 6$ ,  $c(D) = 3$ ,  $c(E) = 1 + 4 = 5$ ,

rezultă că următorul nod expandat este nodul D. Nodul proaspăt expandat este eliminat din lista nodurilor active. În continuare, pentru acest nod se generează toți fiii (nodurile J și K din **Figura 15.5** sunt doar o parte dintre fiii nodului expandat, la ceilalți renunțându-se din lipsă de spațiu), care devin noduri active și sunt adăugați în lista nodurilor active (aceasta va conține nodurile B, C, E, J și K). Pentru nodurile active existente se calculează din nou valorile funcției *c*, obținându-se valoarea minimă pentru nodul J, care devine noul nod expandat, fiind eliminat din lista nodurilor active. Procedul se aplică într-un mod similar, până când se obține nodul răspuns, dacă problema are soluție. În final, în cazul în care problema are soluție, se obține secvența de mutări care a dus la obținerea nodului răspuns. În cazul nostru, această secvență este A, D, J, M.

Lipsa spațiului a făcut ca arborele de stări asociat configurației inițiale să nu conțină toate mutările posibile.

În final, prezentăm implementarea Java a problemei puzzle-ului cu 15 elemente:

---

```

1 import java.util.*;
2 import io.Reader;
3
4 /**
5  * Rezolvarea jocului de Puzzle cu 15 elemente.
6  */
7 public class Puzzle
8 {
9     //pastreaza valorile functiei f ptr. fiecare nod activ
10    public static Vector valoriF = new Vector();
11
12    //distanța de la radacina la un nod
13    public static int distanta = 0;
14
15    public static void main(String[] args)
16    {
17        System.out.println("Introduceti configuratia initiala " +
18            "(ptr. spatiul gol tastati valoarea 16): ");
19
20        int n = 4;
21        int[][] x = new int[n][n];
22
23        obtineConfiguratiaInitiala(x);
24
25        if (existaSolutie(x))
26        {
27            cautaSolutie(x);
28        }
29    }

```

```

29     else
30     {
31         System.out.println("NU exista solutii!!!");
32     }
33 }
34
35 /** Citeste configuratia initiala a jocului. */
36 public static void obtineConfiguratiaInitiala(int[][] x)
37 {
38     for (int i = 0; i < x.length; i++)
39     {
40         for (int j = 0; j < x.length; j++)
41         {
42             System.out.print("x[" + i +
43                 "]" + j + "]=");
44             x[i][j] = Reader.readInt();
45         }
46     }
47 }
48
49 /** Determina daca jocul are solutie. */
50 public static boolean existaSolutie(int[][] x)
51 {
52     int s = 0;
53     for (int i = 1; i <= 16; i++)
54     {
55         s += less(x, i);
56     }
57
58     s += determinaPozSpatiu(x);
59
60     if (s % 2 == 0) return true;
61     else return false;
62 }
63
64 /** Functia position. */
65 public static int position(int[][] x, int el)
66 {
67     for (int i = 0; i < x.length; i++)
68     {
69         for (int j = 0; j < x.length; j++)
70         {
71             if (el == x[i][j])
72             {
73                 return i * x.length + j + 1;
74             }
75         }
76     }
77
78     return 0;

```

```

79  }
80
81  /** Functia less. */
82  public static int less(int[][] x, int i)
83  {
84      int nr = 0;
85
86      for (int j = 1; j < i; j++)
87      {
88          if (position(x, j) > position(x, i))
89          {
90              nr++;
91          }
92      }
93
94      return nr;
95  }
96
97  /** Determina pozitia spatiului pe grila de joc. */
98  public static int determinaPozSpatiu(int[][] x)
99  {
100     for (int i = 0; i < x.length; i++)
101     {
102         for (int j = 0; j < x.length; j++)
103         {
104             // 4 * 4 = 16 = spatiu liber
105             if (x[i][j] == x.length * x.length)
106             {
107                 if ((i % 2 == 0 && j % 2 == 1) ||
108                     (i % 2 == 1 && j % 2 == 0))
109                 {
110                     return 1;
111                 }
112                 else
113                 {
114                     return 0;
115                 }
116             }
117         }
118     }
119
120     return 0;
121 }
122
123 /** Cauta solutia jocului pe baza configuratiei initiale. */
124 public static void cautaSolutie(int[][] x)
125 {
126     // lista nodurilor active
127     Vector noduriActive = new Vector();
128     // nodul expandat (implicit este configuratia initiala)

```

```

129     int [][] nodExpandat = x;
130
131     //pastreaza succesiunea de mutari efectuate
132     Vector mutari = new Vector();
133
134     for (; ; )
135     {
136         distanta++;
137         noduriActive = determinaNoduriActive(nodExpandat,
138                                             noduriActive);
139
140         if (noduriActive.size() == 0) return;
141
142         nodExpandat = determinaNodExpandat(noduriActive);
143
144         mutari.addElement(nodExpandat);
145         if (gc(nodExpandat) == 0)
146         {
147             afiseazaSolutie(mutari);
148             return;
149         }
150         int minPos = noduriActive.indexOf(nodExpandat);
151         noduriActive.removeElementAt(minPos);
152         valoriF.removeElementAt(minPos);
153     }
154 }
155
156 /**
157  * Determina nodurile active din multimea carora se
158  * va alege nodul expandat.
159  */
160 public static Vector determinaNoduriActive(int [][] x,
161                                           Vector noduriActive)
162 {
163     //cauta spatiul gol in interiorul configuratiei
164     int il = 0;
165     int jl = 0;
166
167     for (int i = 0; i < x.length; i++)
168     {
169         for (int j = 0; j < x.length; j++)
170         {
171             if (x[i][j] == x.length * x.length)
172             {
173                 il = i;
174                 jl = j;
175             }
176         }
177     }
178 }

```

```

179     if (i1 != 0) //mutare SUS posibila
180     {
181         int [][] xUp = new int[x.length][x.length];
182
183         copie(x, xUp);
184
185         int aux = xUp[i1 - 1][j1];
186         xUp[i1 - 1][j1] = xUp[i1][j1];
187         xUp[i1][j1] = aux;
188
189         noduriActive.addElement(xUp);
190         valoriF.addElement(new Integer(distanta));
191     }
192
193     if (i1 != x.length - 1) //mutare JOS posibila
194     {
195         int [][] xDown = new int[x.length][x.length];
196
197         copie(x, xDown);
198
199         int aux = xDown[i1][j1];
200         xDown[i1][j1] = xDown[i1 + 1][j1];
201         xDown[i1 + 1][j1] = aux;
202
203         noduriActive.addElement(xDown);
204         valoriF.addElement(new Integer(distanta));
205     }
206
207     if (j1 != 0) //mutare STANGA posibila
208     {
209         int [][] xLeft = new int[x.length][x.length];
210
211         copie(x, xLeft);
212
213         int aux = xLeft[i1][j1 - 1];
214         xLeft[i1][j1 - 1] = xLeft[i1][j1];
215         xLeft[i1][j1] = aux;
216
217         noduriActive.addElement(xLeft);
218         valoriF.addElement(new Integer(distanta));
219     }
220
221     if (j1 != x.length - 1) //mutare DREAPTA posibila
222     {
223         int [][] xRight = new int[x.length][x.length];
224
225         copie(x, xRight);
226
227         int aux = xRight[i1][j1];
228         xRight[i1][j1] = xRight[i1][j1 + 1];

```

```

229     xRight[i1][j1 + 1] = aux;
230
231     noduriActive.addElement(xRight);
232     valoriF.addElement(new Integer(distanta));
233 }
234 return noduriActive;
235 }
236
237 /**
238  * Metoda utila de copiere a valorilor dintr-un sir sursa
239  * intr-un sir destinatie.
240  */
241 public static void copie(int[][] src, int[][] dest)
242 {
243     for (int i = 0; i < src.length; i++)
244     {
245         for (int j = 0; j < src.length; j++)
246         {
247             dest[i][j] = src[i][j];
248         }
249     }
250 }
251
252 /** Determina nodul expandat pe baza nodurilor active. */
253 public static int[][] determinaNodExpandat(Vector noduriActive)
254 {
255     int minPos = 0;
256     int min = Integer.MAX_VALUE;
257
258     for (int i = 0; i < noduriActive.size(); i++)
259     {
260         int[][] el = (int[][]) noduriActive.elementAt(i);
261
262         if (cc(el, i) < min)
263         {
264             min = cc(el, i);
265             minPos = i;
266         }
267     }
268
269     return (int[][]) noduriActive.elementAt(minPos);
270 }
271
272 /** Calculul functiei c. */
273 public static int cc(int[][] x, int j)
274 {
275     return gc(x) + f(j);
276 }
277
278 /** Calculul functiei g. */

```

```

279 public static int gc(int [][] x)
280 {
281     int nr = 0;
282
283     for (int i = 0; i < x.length; i++)
284     {
285         for (int j = 0; j < x.length; j++)
286         {
287             if (x[i][j] != i * x.length + j + 1 &&
288                 x[i][j] != 16)
289             {
290                 nr++;
291             }
292         }
293     }
294
295     return nr;
296 }
297
298 /** Calculul functiei f. */
299 public static int f(int j)
300 {
301     return ((Integer) valoriF.elementAt(j)).intValue();
302 }
303
304 /** Afisarea solutiei obtinute. */
305 public static void afiseazaSolutie(Vector mutari)
306 {
307     System.out.println("Mutarile efectuate: ");
308     for (int i = 0; i < mutari.size(); i++)
309     {
310         int [][] m = (int [][]) mutari.elementAt(i);
311
312         for (int j = 0; j < m.length; j++)
313         {
314             for (int k = 0; k < m.length; k++)
315             {
316                 if (m[j][k] == 16) // spatiu
317                 {
318                     System.out.print("   ");
319                 }
320                 else if (m[j][k] < 10)
321                 {
322                     System.out.print("  " + m[j][k]);
323                 }
324                 else
325                 {
326                     System.out.print(" " + m[j][k]);
327                 }
328             }

```



```

329         System.out.println();
330     }
331     System.out.println();
332 }
333 }
334 }

```

## Rezumat

Capitolul de față a prezentat metoda *Branch & bound* de elaborare a algoritmilor. Asemănătoare metodei *Backtracking*, această metodă este mai rar utilizată decât *Backtracking*-ul, de aceea a fost prezentată în mai puține detalii. S-a insistat însă asupra mecanismului ei de funcționare, care se bazează pe o funcție de cost asociată nodurilor din arborele de stări. Funcția de cost trebuie aleasă în funcție de problema care trebuie rezolvată. Problema puzzle-ului cu 15 elemente este un bun exemplu pentru aprofundarea cunoștințelor de *Branch and bound*.

## Noțiuni fundamentale

**arbore de stări:** arbore oarecare generat de aplicarea mutărilor permise de problemă unei configurații inițiale.

**funcție de cost:** funcție specială care diferă de la problemă la problemă și este asociată unui nod din arborele de stări. Cu ajutorul ei se elimină subarborii care nu duc la rezultatul dorit, obținându-se o reducere semnificativă a timpului de aflare a soluției.

**nod activ:** nod care a fost obținut prin expandarea unui alt nod și care nu a fost încă, la rândul lui, expandat.

**nod expandat:** nodul curent, pentru care au fost generați fiii (s-au realizat mutările permise de problemă)

**nod răspuns:** configurația finală la care trebuie să se ajungă prin efectuarea mutărilor permise asupra configurației inițiale.

## Exerciții

### Teorie

1. Fie  $S = x + \sum_{i=1}^{16} less(i)$  pentru o configurație oarecare a problemei puzzle cu 15 elemente. Demonstrați că  $S \bmod 2$  este invariant pentru

orice stare obținută din starea inițială. Cu alte cuvinte, dacă  $S$  este impară, atunci  $S$  rămâne impară pentru orice secvență de mutări legală, iar dacă este pară, atunci rămâne pară.

2. Folosind observația de la exercițiul precedent, demonstrați teorema 15.2.1.

### În practică

1. Rezolvați problema discretă a rucsacului folosind o abordare *Branch and bound*. Reamintim că în problema discretă a rucsacului se dau un număr de  $n$  produse, fiecare cu greutatea  $w_i$  și costul  $p_i$ , și un număr pozitiv  $M$  ce reprezintă capacitatea rucsacului. Se cere să se aleagă o submulțime de produse astfel încât  $\sum_1^n w_i x_i \leq M$  și  $\sum_1^n p_i x_i$  este maxim, unde  $x$  este un vector de dimensiune  $n$  ale cărui elemente se află în intervalul  $[0, 1]$ . Altfel spus, este posibil ca dintr-un produs să se aleagă doar o anumită parte și nu întregul.
2. Rezolvați problema comis-voiajorului folosind metoda *Branch and bound*. Reamintim că problema comis-voiajorului are următoarea definiție: un comis voiajor trebuie să treacă prin toate orașele dintr-un județ, doar o singură dată, astfel încât costul drumului realizat să fie minim.

3. Se dă următoarea problemă de optimizare:

$$\max 3x_1 + x_2 + x_3 + 2x_4 + 2x_5 + 3x_6$$

având restricțiile:

$$\begin{array}{rcccccccl} x_1 & +x_2 & & & & & & +y_1 = 1 \\ x_1 & & +x_3 & & & & & +y_2 = 1 \\ & x_2 & +x_3 & & & & & +y_3 = 1 \\ & x_2 & & +x_4 & & & & +y_4 = 1 \\ & & x_3 & & +x_5 & & & +y_5 = 1 \\ & & & x_4 & +x_5 & & & +y_6 = 1 \\ & & & x_4 & & +x_6 & +y_7 = 1 \\ & & & & x_5 & +x_6 & +y_8 = 1 \end{array}$$

$$x_i \in \{0, 1\}, y_i \geq 0 \quad i = 1, 2, \dots, 6$$

Observați că spațiul de căutare al problemei este finit ( $2^6$ ). Să se găsească soluția optimă a problemei utilizând mai întâi o strategie de tip Backtracking, iar apoi o strategie tip Branch & bound.

## 16. Metode de elaborare a algoritmilor (sinteză)

Punctul de convergență al artei și științei este metoda.

---

Edward Bulwer-Lytton

Vom prezenta o scurtă sinteză a metodelor de elaborare a algoritmilor, care poate fi utilizată ca o referință rapidă în clasificarea problemelor de algoritmică.

### 16.1 Backtracking

Metoda backtracking se poate aplica problemelor a căror soluție se scrie sub formă de șir, cu fiecare componentă a șirului aparținând unei mulțimi finite. Soluția se construiește incremental, începând cu prima componentă și mergând către ultima, cu eventuale reveniri la componentele anterioare. Fiecare componentă trebuie să respecte condițiile de continuare, care sunt derivate din condițiile interne.

Complexitatea algoritmilor backtracking este în general exponențială. Cu cât condiția de continuare este mai bine aleasă (elimină cât mai multe soluții parțiale care nu pot conduce la o soluție), cu atât soluția va fi mai eficientă.

### 16.2 Divide et impera

Divide et impera se aplică problemelor care se pot descompune în subprobleme, astfel încât soluția problemei originale să se poată obține ușor din soluțiile subproblemelor. Subproblemele se împart la rândul lor în subprobleme până

când se ajunge la subprobleme triviale, care admit soluție imediată. Exprimarea rezolvării este recursivă, ca și algoritmi care utilizează această metodă.

Timpul de calcul este, în general, de forma  $n^k \log n$ . Cele mai eficiente metode de sortare (Mergesort, Quicksort) se bazează pe această metodă.

## 16.3 Greedy

Metoda Greedy se aplică problemelor de optimizare care respectă principiul optimalității (substructură optimă) și principiul alegerii Greedy. Principiul optimalității ne asigură că o soluție optimă cuprinde subsoluții optime. Principiul alegerii Greedy ne asigură că alegând la fiecare pas optimul local, se obține optimul global. Există multe situații în care metoda Greedy se aplică problemelor care nu respectă principiul alegerii Greedy (de exemplu, problema discretă a rucsacului), caz în care soluția dată de metoda Greedy nu va mai fi întotdeauna cea optimă.

Datorită simplității strategiei, soluțiile Greedy au cel mai adesea complexitate polinomială.

## 16.4 Programare dinamică

Programarea dinamică se aplică problemelor de optimizare care respectă principiul optimalității. Reiese imediat că programarea dinamică se aplică unei clase mai largi de probleme decât metoda Greedy. Problemele de programare dinamică se rezolvă prin construirea soluției problemei pe baza soluțiilor optime ale subproblemelor. Din acest punct de vedere, programarea dinamică se află între Greedy, care ia în considerare o singură subproblemă (cea optimă local), și Backtracking, care generează exhaustiv toate soluțiile.

Complexitatea algoritmilor de programare dinamică este în general polinomială (vezi toate problemele prezentate în capitolul dedicat metodei Greedy), dar poate fi și exponențială (problema comis-voiajorului).

Abordarea rezolvării este bottom-up, ceea ce înseamnă că se rezolvă mai întâi problemele de dimensiune mai mică, pe baza lor cele mai mari, până când se ajunge la soluția problemei inițiale.

## 16.5 Branch & bound

Branch & bound este o variantă a metodei backtracking, în care alegerea următorului element nu se face pur și simplu aleator (sau în ordine crescătoare-

re) ci se urmărește alegerea variantelor care sunt mai promițătoare în vederea obținerii unei soluții. Aceasta înseamnă că fiecare posibilă variantă este evaluată după anumite criterii specifice fiecărei probleme în parte, alegându-se la fiecare pas varianta evaluată a fi optimă.

**Tabel sinteză:**

Tabelul următor prezintă o sinteză a informațiilor din acest paragraf.

Metoda	Condiții de aplicare	Timp de lucru (în general)	Construire soluție
Backtracking	Șir cu elemente luând valori în mulțimi finite	exponențial	incremental
Divide et Impera	Soluția se poate construi pe baza soluțiilor subproblemelor	$n^k * \log n$	top-down
Greedy	Probleme de optimizare care respectă principiul optimalității și al alegerii Greedy	polinomial	incremental
Programare dinamică	Probleme de optimizare care respectă principiul optimalității	polinomial	bottom-up
Branch & bound	Probleme de optimizare dificile în care aplicarea celorlalte metode nu este adecvată.	diferă funcție de problemă	parcursare least-cost



# Bibliografie

- [Andonie] R. Andonie, I. Gârbacea - Algoritmi și Calculabilitate, Computer Press Agora, 1995
- [Balanescu] T.Bălănescu - Metoda Backtracking, Gazeta de informatică, nr 2/1993, pag. 9-18
- [Boian] F.M. Boian - De la aritmetică la calculatoare, Presa Universitară Clujeană, 1996, ISBN 973-97535-5-8
- [Cormen] T.H. Cormen, C.E. Leiserson, R.R. Rivest - Introducere în Algoritmi, Computer Press Agora, 2000, ISBN 973-97534-3-4
- [Danciu] D.Danciu, S.Dumitrescu - Algoritmă și Programare, Curs și Probleme de Seminar, Reprografia Univ. Transilvania, 2002
- [Eckel] B. Eckel - Thinking in Java, <http://www.mindview.net/Books/TIJ/>
- [Horowitz] E. Horowitz, S. Sahni - Fundamentals of Computer Algorithms, Computer Science Press, ISBN 3-540-12035-1
- [Norton] P. Norton, W. Stanek - Peter Norton's Guide to Java Programming, Sams.net Publishing, 1996, ISBN 1-57521-088-6
- [Roman] Ed Roman - Mastering Enterprise JavaBeans, Wiley Computer Publishing, 1999, ISBN 0-471-33229-1
- [Sorin] T.Sorin - Tehnici de programare, manual pentru clasa a X-a, L&S Infomat, 1996
- [Tomescu] I. Tomescu - Data Structures, Bucharest University Press, Bucharest, 1997

- [Vanderburg] G. Vanderburg - Tricks of the Java Programming Gurus,  
Sams.net Publishing, 1996, ISBN 1-57521-102-5
- [Weiss] M.A. Weiss - Data Structures and Problem Solving Using Java,  
Addison-Wesley, 1998, ISBN 0-201-54991-3



# Index

$\Omega(f)$ , 21  
 $\Theta(f)$ , 21

acces secvențial, 56  
ActiveX, 7  
alegere optimă, 167  
algoritm, 15  
algoritm exponențial, 36  
algoritm liniar, 36  
algoritm polinomial, 36  
algoritm recursiv, 25  
apel recursiv, 139, 142  
arbore, 104, 237  
arbore binar, 104  
arbore binar de căutare, 69  
arbore de stări, 253  
autoapelare, 138  
  
backtracking, 12, 108, 130, 255  
BFS, 238  
branch and bound, 12, 236, 256  
  
căutare LC, 239  
căutare binară, 162  
candidați, 173  
celulă, 240  
cercetări operaționale, 187  
coadă, 51  
coadă de prioritate, 93  
complexitate, 15  
condiție de terminare, 139, 162

condiții de continuare, 110, 130  
condiții interne, 109, 110, 130  
configurație, 112, 237  
configurație finală, 116, 130, 237  
configurație inițială, 113, 130, 237  
configurație soluție, 113  
corectitudinea algoritmilor, 169  
cost, 239

dequeue, 51  
DFS, 238  
distanță, 220  
divide et impera, 12, 255

E-node, 238  
enqueue, 51  
enterprise, 7  
expandare, 238  
expresie aritmetică, 157

fezabil, 174  
FIFO, 238  
forma poloneză, 157  
formule de recurență, 210  
frunză, 104, 237, 238  
funcție de selecție, 174, 183  
funcție obiectiv, 174  
funcție de cost, 239, 253

getFront, 51  
greedy, 12, 256

- GUI, 7
- hash, 104
- hashtable, 83
- inteligență artificială, 238
- interclasare, 180
- internet, 7
- iterator, 57, 104
- Java 2 Platform, 7
- LC (Least Cost), 239
- LIFO, 238
- listă înlănțuită, 55
- lungime externă ponderată, 181
- mergesort, 149, 162
- metodă recursivă, 162
- mutări, 237
- nod, 55, 238
- nod încă neexpandat, 238
- nod activ, 238, 253
- nod curent, 238
- nod E, 238
- nod expandat, 238, 253
- nod răspuns, 238, 253
- nod terminal, 238
- notație asimptotică, 18
- numere catalane, 194
- $O(f)$ , 19
- operații recursive, 143
- operand, 157
- operator, 157
- optim global, 167, 176
- optim local, 176
- optimalitate, 201
- optimizare, 167, 186
- ordonare, 149
- 262
- pager, 7
- partiționare, 152
- PDA, 7
- pivot, 152
- principiul alegerii greedy, 168
- principiul invarianței, 19
- principiul optimalității, 168, 183, 186, 226
- probare liniară, 91
- probare pătratică, 91
- problemă de optimizare, 183, 226
- programare dinamică, 12, 226, 256
- proprietatea de alegere greedy, 173, 183
- quicksort, 152, 162
- rădăcină, 237
- Reader, 120
- recurență, 138, 211
- recursivitate, 137
- relație de recurență, 221
- relații Moivre, 28
- Shockwave, 7
- soluție optimă, 226
- soluție optimală, 237
- sortare, 137
- spațiul soluțiilor posibile, 109
- spațiul stărilor, 236
- stivă, 45
- stiva programului, 142
- strategie euristică, 173
- strategie greedy, 167
- structură de date, 40
- subproblemă, 137, 187, 226
- substructură optimă, 168, 173, 176, 183, 186, 216
- succesor, 238
- Sun Microsystems, 7

suprapunere de apeluri, 162

tabel, 188, 189

tabelă de repartizare, 83

teoria grafurilor, 167

triunghiul lui Pascal, 192

turnurile din Hanoi, 24

valori consumate, 111

web, 7